The ZeroMQ Guide - for Lua Developers

The ZeroMQ Guide - for Lua Developers

Dedication

By Pieter Hintjens

With thanks to the hundred or so people who contributed examples in two dozen programming languages, who helped with suggestions and fixes, and who kept pushing for more examples of how to connect your code.

Thanks to Bill Desmarais, Brian Dorsey, Daniel Lin, Eric Desgranges, Gonzalo Diethelm, Guido Goldstein, Hunter Ford, Kamil Shakirov, Martin Sustrik, Mike Castleman, Naveen Chawla, Nicola Peduzzi, Oliver Smith, Olivier Chamoux, Peter Alexander, Pierre Rouleau, Randy Dryburgh, John Unwin, Alex Thomas, Mihail Minkov, Jeremy Avnet, Michael Compton, Kamil Kisiel, Mark Kharitonov, Guillaume Aubert, Ian Barber, Mike Sheridan, Faruk Akgul, Oleg Sidorov, Lev Givon, Allister MacLeod, Alexander D'Archangel, Andreas Hoelzlwimmer, Han Holl, Robert G. Jakabosky, Felipe Cruz, Marcus McCurdy, Mikhail Kulemin, Dr. Gergö Érdi, Pavel Zhukov, Alexander Else, Giovanni Ruggiero, Rick "Technoweenie", Daniel Lundin, Dave Hoover, Simon Jefford, Benjamin Peterson, Justin Case, Devon Weller, Richard Smith, Alexander Morland, Wadim Grasza, Michael Jakl, Uwe Dauernheim, Sebastian Nowicki, Simone Deponti, Aaron Raddon, Dan Colish, Markus Schirp, Benoit Larroque, Jonathan Palardy, Isaiah Peng, Arkadiusz Orzechowski, Umut Aydin, Matthew Horsfall, Jeremy W. Sherman, Eric Pugh, Tyler Sellon, John E. Vincent, Pavel Mitin, Min RK, Igor Wiedler, Olof Åkesson, Patrick Lucas, Heow Goodman, Senthil Palanisami, John Gallagher, Tomas Roos, Stephen McQuay, Erik Allik, Arnaud Cogoluègnes, Rob Gagnon, Dan Williams, Edward Smith, James Tucker, Kristian Kristensen, Vadim Shalts, Martin Trojer, Tom van Leeuwen, Pandya Hiten, Harm Aarts, Marc Harter, Iskren Ivov Chernev, Jay Han, Sonia Hamilton, and Zed Shaw.

Thanks to Stathis Sideris for Ditaa (http://www.ditaa.org), which I used for the diagrams.

Please use the issue tracker (https://github.com/imatix/zguide2/issues) for all comments and errata. This version covers the latest stable release of ØMQ (3.2) and was published on Tue 30 October, 2012. If you are using older versions of ØMQ then some of the examples and explanations won't be accurate.

The Guide is originally in C (http://zguide.zeromq.org/page:all), but also in PHP (http://zguide.zeromq.org/php:all), Python (http://zguide.zeromq.org/py:all), Lua (http://zguide.zeromq.org/lua:all), and Haxe (http://zguide.zeromq.org/hx:all). We've also translated most of the examples into C++, C#, CL, Erlang, F#, Felix, Haskell, Java, Objective-C, Ruby, Ada, Basic, Clojure, Go, Haxe, Node.js, ooc, Perl, and Scala.

Table of Contents

List of Figures

Chapter 1. Basic Stuff

1.1. Fixing the World

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 leading software engineers like Fred Brooks believed there was no "Silver Bullet" (http://en.wikipedia.org/wiki/No_Silver_Bullet) to "promise even one order of magnitude of improvement in productivity, reliability, or simplicity".

Brooks missed free and open source software, which 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 (http://swsi.info). 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 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.

1.2. ØMQ in a Hundred Words

ØMQ (ZeroMQ, ØMQ, zmq) looks like an embeddable networking library but acts like a concurrency framework. It gives you sockets that carry atomic 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 (http://www.imatix.com) and is LGPLv3 open source.

1.3. Some Assumptions

We assume you are using the latest 3.2 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.

1.4. Getting the Examples

The Guide examples live in the Guide's git repository (https://github.com/imatix/zguide2). The simplest way to get all the examples is to clone this repository:

```
git clone --depth=1 git://github.com/imatix/zguide2.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 (http://zguide2.zeromq.org/main:translate). This is how the Guide became so useful, thanks to the work of many people. All examples are licensed under MIT/X11.

1.5. Ask and Ye Shall Receive

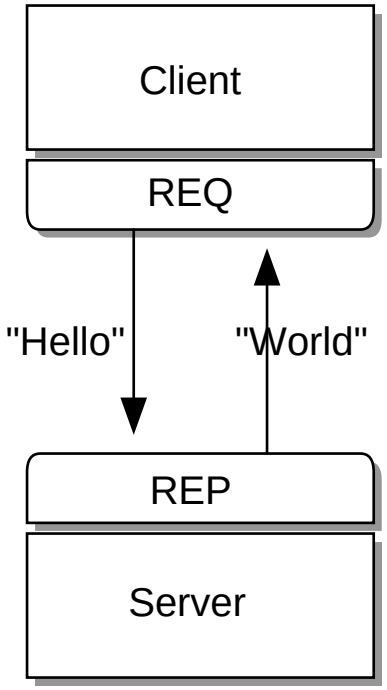
So let's start with some code. We start of course with a Hello World example. We'll make a client and a server. The client sends "Hello" to the server, which replies with "World"(Figure 1-1). Here's the server in C, which opens a ØMQ socket on port 5555, reads requests on it, and replies with "World" to each request:

Example 1-1. Hello World server (hwserver.c)

```
//
//  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_ctx_new ();

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

    while (true) {
        //  Wait for next request from client
        zmq_msg_t request;
        zmq_msg_init (&request);
        zmq_msg_recv (&request, responder, 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_msg_send (&reply, responder, 0);
        zmq_msg_close (&reply);
    }
    //  We never get here but if we did, this would be how we end
    zmq_close (responder);
    zmq_ctx_destroy (context);
    return 0;
}
```

Figure 1-1. Request-Reply



The REQ-REP socket pair is lockstep. The client does zmq_msg_send[3] and then zmq_msg_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 result in a return code of -1 from the send or recv call. Similarly the service does zmq_msg_recv[3] and then zmq_msg_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++:

Example 1-2. Hello World server (hwserver.cpp)

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

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:

Example 1-3. Hello World server (hwserver.php)

```
<?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");
```

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

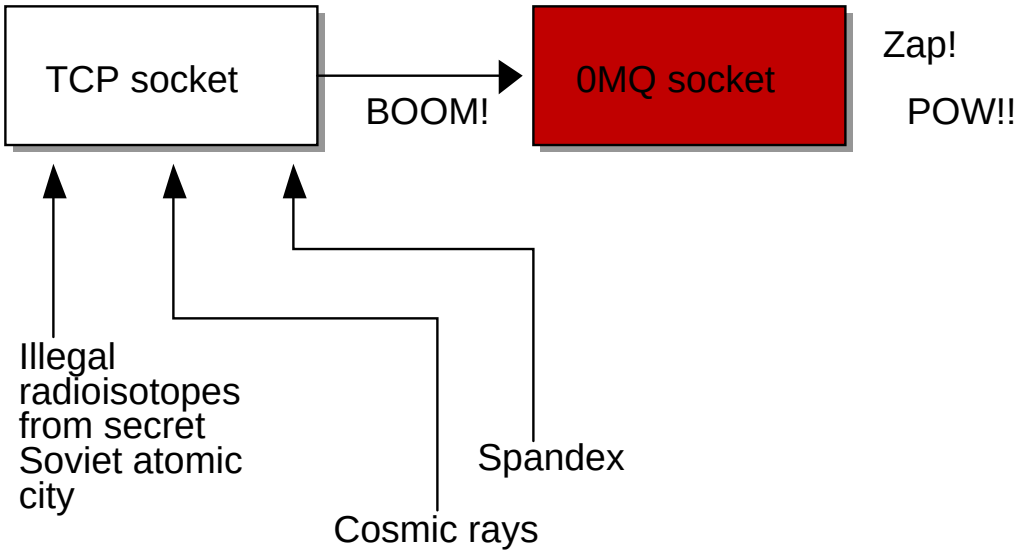
Example 1-4. Hello World client (hwclient.lua)

```
--
--  Hello World client
--  Connects REQ socket to tcp://localhost:5555
--  Sends "Hello" to server, expects "World" back
--
--  Author: Robert G. Jakabosky <bobby@sharedrealm.com>
--
require"zmq"

local context = zmq.init(1)

--  Socket to talk to server
print("Connecting to hello world server...")
local socket = context:socket(zmq.REQ)
socket:connect("tcp://localhost:5555")

for n=1,10 do
    print("Sending Hello " .. n .. " ...")
    socket:send("Hello")

    local reply = socket:recv()
    print("Received World " ..  n .. " [" .. reply .. "]")
end
socket:close()
context:term()
```

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(Figure 1-2). Yes, ØMQ sockets are the world-saving superheroes of the networking world.

Figure 1-2. 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.

If you kill the server (Ctrl-C) and restart it, the client won't recover properly. Recovering from crashing processes isn't quite that easy. Making a reliable request-reply flow is complex enough that I won't cover it until Chapter Four.

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.

1.6. A Minor Note on Strings

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:

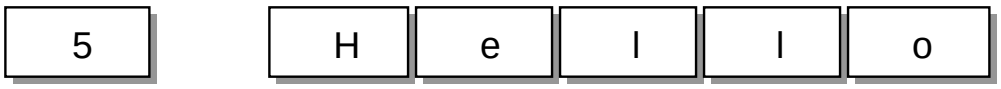
```
zmq_msg_init_data (&request, "Hello", 6, NULL, NULL);
```

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 a length (one byte for shorter strings) and the string contents, as individual characters(Figure 1-3).

Figure 1-3. 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);
    int size = zmq_msg_recv (&message, socket, 0);
    if (size == -1)
        return NULL;
    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 (https://github.com/imatix/zguide2/blob/master/examples/C/zhelpers.h).

1.7. Version Reporting

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:

Example 1-5. ØMQ version reporting (version.lua)

```
--
--  Report 0MQ version
--
--  Author: Robert G. Jakabosky <bobby@sharedrealm.com>
--
require"zmq"

print("Current 0MQ version is " .. table.concat(zmq.version(), '.'))
```

1.8. Getting the Message Out

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 zip code, temperature, and relative humidity. We'll generate random values, just like the real weather stations do.

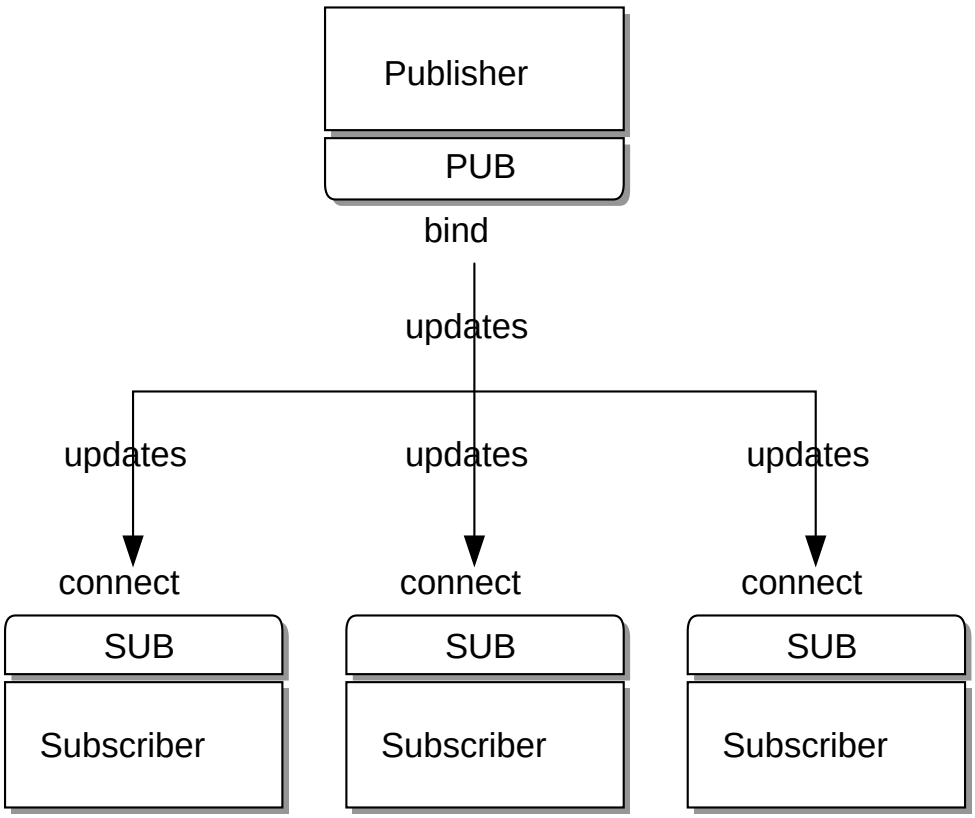
Here's the server. We'll use port 5556 for this application:

Example 1-6. Weather update server (wuserver.lua)

```
--
--  Weather update server
--  Binds PUB socket to tcp://*:5556
--  Publishes random weather updates
--
--  Author: Robert G. Jakabosky <bobby@sharedrealm.com>
--
require"zmq"

--  Prepare our context and publisher
local context = zmq.init(1)
local publisher = context:socket(zmq.PUB)
publisher:bind("tcp://*:5556")
publisher:bind("ipc://weather.ipc")

--  Initialize random number generator
math.randomseed(os.time())
while (1) do
    --  Get values that will fool the boss
    local zipcode, temperature, relhumidity
    zipcode     = math.random(0, 99999)
    temperature = math.random(-80, 135)
    relhumidity = math.random(10, 60)

    --  Send message to all subscribers
    publisher:send(string.format("%05d %d %d", zipcode, temperature, relhumidity))
end
publisher:close()
context:term()
```

There's no start, and no end to this stream of updates, it's like a never ending broadcast(Figure 1-4).

Figure 1-4. Publish-Subscribe



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

Example 1-7. Weather update client (wuclient.lua)

```
--
--  Weather update client
--  Connects SUB socket to tcp://localhost:5556
--  Collects weather updates and finds avg temp in zipcode
--
--  Author: Robert G. Jakabosky <bobby@sharedrealm.com>
--
require"zmq"

local context = zmq.init(1)

--  Socket to talk to server
print("Collecting updates from weather server...")
```

```
local subscriber = context:socket(zmq.SUB)
subscriber:connect(arg[2] or "tcp://localhost:5556")

--  Subscribe to zipcode, default is NYC, 10001
local filter = arg[1] or "10001 "
subscriber:setopt(zmq.SUBSCRIBE, filter)

--  Process 100 updates
local update_nbr = 0
local total_temp = 0
for n=1,100 do
    local message = subscriber:recv()
    local zipcode, temperature, relhumidity = message:match("([%d-]*) ([%d-]*) ([%d-]*)")
    total_temp = total_temp + temperature
    update_nbr = update_nbr + 1
end
print(string.format("Average temperature for zipcode '%s' was %dF, total = %d",
    filter, (total_temp / update_nbr), total_temp))

subscriber:close()
context:term()
```

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_msg_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_msg_send[3] as often as it needs to, but must not do zmq_msg_recv[3] on a PUB socket.

In theory with ØMQ sockets, it does not matter which end connects, and which end binds. However in practice there are undocumented differences that I'll come to later. For now, bind the PUB and connect the SUB, unless your network design makes that impossible.

There is one more 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 those 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 connect to more than one publisher, using one 'connect' call each time. Data will then arrive and be interleaved ("fair-queued") 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 laptop, which is an 2011-era Intel I7, fast but nothing special:

```
ph@nb201103:~/work/git/zguide/examples/c$ time wuclient
Collecting updates from weather server...
```

```
Average temperature for zipcode '10001 ' was 28F

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

1.9. Divide and Conquer

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(Figure 1-5):

- 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.

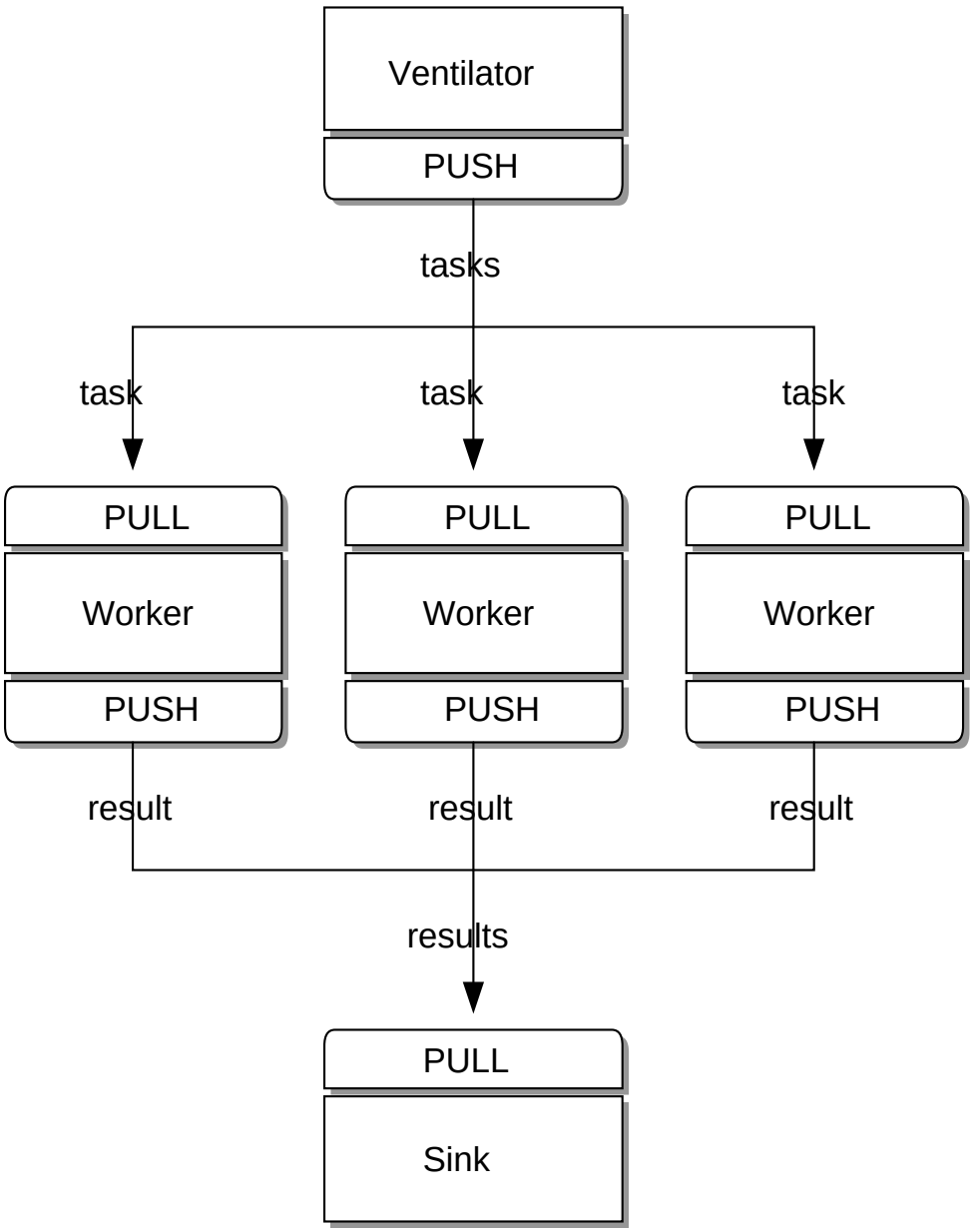
In reality, workers run on superfast boxes, perhaps using GPUs (graphic processing units) to do the hard maths. Here is the ventilator. It generates 100 tasks, each is a message telling the worker to sleep for some number of milliseconds:

Example 1-8. Parallel task ventilator (taskvent.lua)

```
--
--  Task ventilator
--  Binds PUSH socket to tcp://localhost:5557
--  Sends batch of tasks to workers via that socket
--
--  Author: Robert G. Jakabosky <bobby@sharedrealm.com>
--
require"zmq"
require"zhelpers"

local context = zmq.init(1)

--  Socket to send messages on
local sender = context:socket(zmq.PUSH)
sender:bind("tcp://*:5557")

printf ("Press Enter when the workers are ready: ")
io.read('*l')
printf ("Sending tasks to workers...\n")

--  The first message is "0" and signals start of batch
sender:send("0")

--  Initialize random number generator
math.randomseed(os.time())
```

```
--  Send 100 tasks
local task_nbr
local total_msec = 0     --  Total expected cost in msecs
for task_nbr=0,99 do
    local workload
    --  Random workload from 1 to 100msecs
    workload = randof (100) + 1
    total_msec = total_msec + workload
    local msg = string.format("%d", workload)
    sender:send(msg)
end
printf ("Total expected cost: %d msec\n", total_msec)
s_sleep (1000)              --  Give 0MQ time to deliver

sender:close()
context:term()
```

Figure 1-5. Parallel Pipeline



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

Example 1-9. Parallel task worker (taskwork.lua)

```
--
--  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
--
--  Author: Robert G. Jakabosky <bobby@sharedrealm.com>
--
require"zmq"
require"zhelpers"

local context = zmq.init(1)

--  Socket to receive messages on
local receiver = context:socket(zmq.PULL)
receiver:connect("tcp://localhost:5557")

--  Socket to send messages to
local sender = context:socket(zmq.PUSH)
sender:connect("tcp://localhost:5558")

--  Process tasks forever
while true do
    local msg = receiver:recv()
    --  Simple progress indicator for the viewer
    io.stdout:flush()
    printf("%s.", msg)

    --  Do the work
    s_sleep(tonumber(msg))

    --  Send results to sink
    sender:send("")
end
receiver:close()
sender:close()
context:term()
```

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:

Example 1-10. Parallel task sink (tasksink.lua)

```
--
--  Task sink
--  Binds PULL socket to tcp://localhost:5558
--  Collects results from workers via that socket
--
```

```
--  Author: Robert G. Jakabosky <bobby@sharedrealm.com>
--
require"zmq"
require"zhelpers"
local fmod = math.fmod

--  Prepare our context and socket
local context = zmq.init(1)
local receiver = context:socket(zmq.PULL)
receiver:bind("tcp://*:5558")

--  Wait for start of batch
local msg = receiver:recv()

--  Start our clock now
local start_time = s_clock ()

--  Process 100 confirmations
local task_nbr
for task_nbr=0,99 do
    local msg = receiver:recv()

    if (fmod(task_nbr, 10) == 0) then
        printf (":")
    else
        printf (".")
    end
    io.stdout:flush()
end
--  Calculate and report duration of batch
printf("Total elapsed time: %d msec\n", (s_clock () - start_time))

receiver:close()
context:term()
```

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

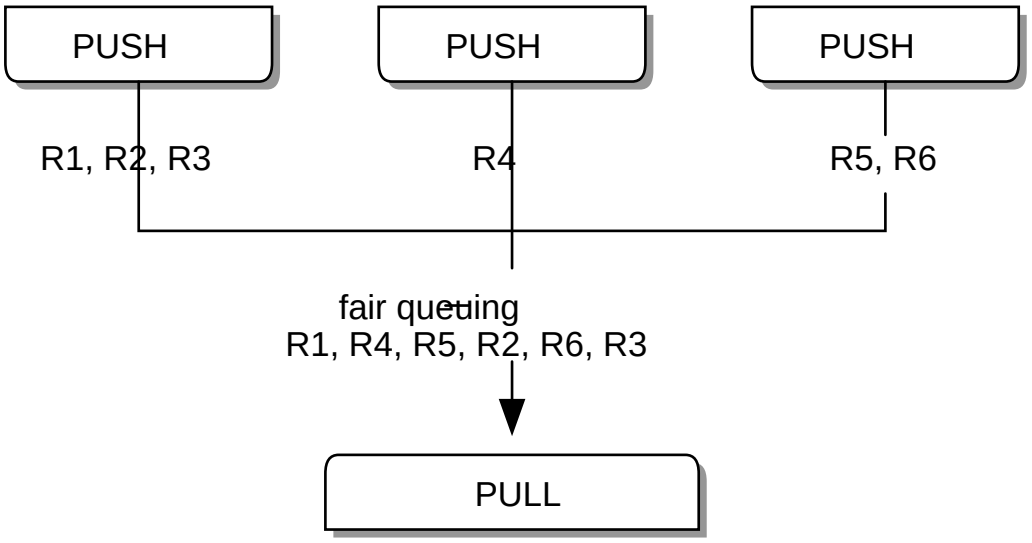
```
#   1 worker
Total elapsed time: 5034 msec
#   2 workers
Total elapsed time: 2421 msec
#   4 workers
Total elapsed time: 1018 msec
```

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 1-6).

Figure 1-6. 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.

1.10. Programming with ØMQ

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 few 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, NULL);

    fprintf(stdout,"WORKER: ready to receive 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_msg_send(pubskt, &topic_msg, ZMQ_SNDMORE);

    zmq_msg_init(&cmd_msg);
    zmq_msg_recv(qskt, &cmd_msg, 0);
```

```
    memcpy(&cmd, zmq_msg_data(&cmd_msg), sizeof(uint8_t));
    zmq_msg_send(pubskt, &cmd_msg, ZMQ_SNDMORE);
    zmq_msg_close(&cmd_msg);

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

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

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

    zmq_msg_init_size(&resp_msg, sizeof(uint8_t));
    memset(zmq_msg_data(&resp_msg), 0, sizeof(uint8_t));
    zmq_msg_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 crashes weirdly. Sockets are not threadsafe. It became legal behavior to migrate sockets from one thread to another in ØMQ/2.1, but this remains dangerous unless you use a "full memory barrier". If you don't know what that means, don't attempt socket migration.

1.11. Getting the Context Right

ØMQ applications always start by creating a *context*, and then using that for creating sockets. In C, it's the zmq_ctx_new[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_ctx_new[3] at the start of your main line code, and one zmq_ctx_destroy[3] at the end.

If you're using the fork() system call, each process needs its own context. If you do zmq_ctx_new[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.

1.12. Making a Clean Exit

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 are 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_ctx_destroy[3] function will hang forever. And even if you close all sockets, zmq_ctx_destroy[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_ctx_destroy[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, you need to shut down each socket that has ongoing requests. The proper way is to set a low LINGER value (1 second), then close the socket. If your language binding doesn't do this for you automatically when you destroy a context, I'd suggest sending a patch.

Finally, destroy the context. This will 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 error, and then set linger on, and close sockets in *that* thread, and exit. Do not destroy the same context twice. The zmq_ctx_destroy 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.

1.13. Why We Needed ØMQ

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.

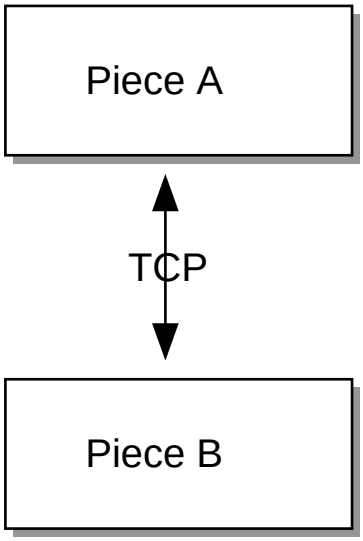
Let's look at the typical problems we face when we start to connect pieces using raw TCP. Any reusable messaging layer would need to solve all or most these:

- 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 (http://hadoop.apache.org/zookeeper/) and read the C API code in src/c/src/zookeeper.c (http://github.com/apache/zookeeper/blob/trunk/src/c/src/zookeeper.c). As I write this, in 2010, the code is 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 1-7. 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?

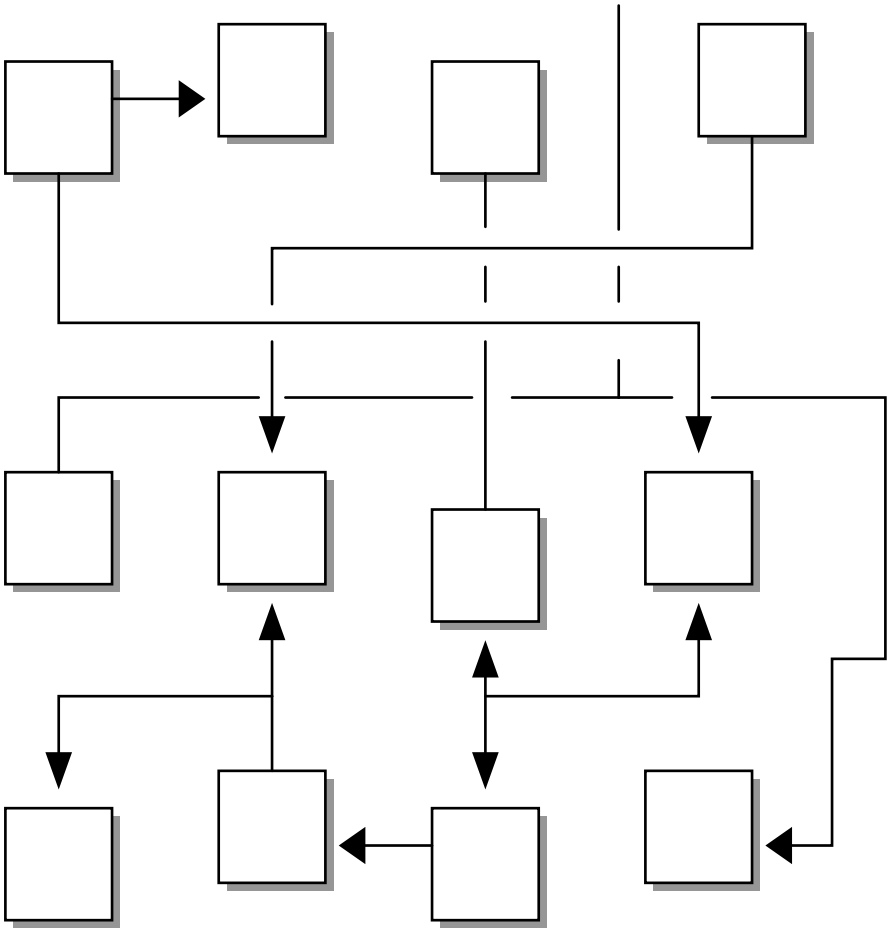
It turns out that building reusable messaging systems is really difficult, which is why few FOSS projects ever tried, and why commercial messaging products are complex, expensive, inflexible, and brittle. In 2006 iMatix designed AMQP (http://www.amqp.org) which started to give FOSS developers perhaps the first reusable recipe for a messaging system. AMQP works better than many other designs but remains relatively complex, expensive, and brittle (http://www.imatix.com/articles:whats-wrong-with-amqp). 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 1-8. 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 concurrent 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 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 create proxies to queue, forward, or capture messages with a single call. Proxies can reduce the interconnection complexity of a network.
- 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 a socket-inspired API on which you do zmq_msg_recv[3] and zmq_msg_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), or two boxes on one network (node is a box) - it's all the same, with no application code changes.

1.14. Socket Scalability

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):

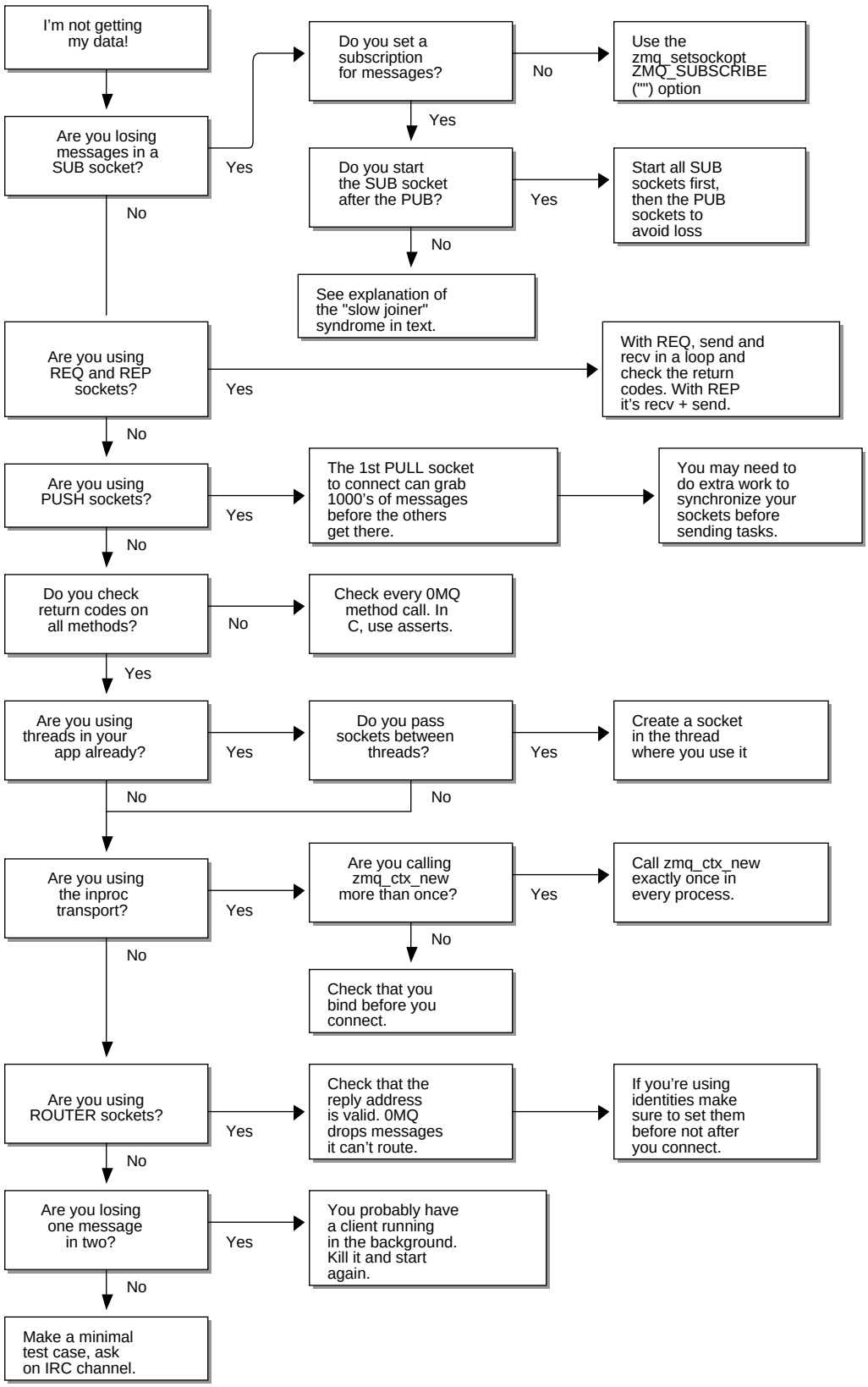
```
  PID USER      PR  NI  VIRT  RES  SHR S %CPU %MEM    TIME+  COMMAND
 7136 ph        20   0 1040m 959m 1156 R  157 12.0  16:25.47 wuserver
 7966 ph        20   0 98608 1804 1372 S   33  0.0   0:03.94 wuclient
 7963 ph        20   0 33116 1748 1372 S   14  0.0   0:00.76 wuclient
 7965 ph        20   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. So the ØMQ socket is acting like a little server, silently accepting client requests and shoving data out to them as fast as the network can handle it. And it's a multithreaded server, squeezing more juice out of your CPU.

1.15. Missing Message Problem Solver

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(Figure 1-9) 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.

Figure 1-9. Missing Message Problem Solver



If you're using ØMQ in a context where failures are expensive, then you want to 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 (http://www.imatix.com/contact) 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.

1.16. Upgrading from ØMQ/2.2 to ØMQ/3.2

In early 2012, ØMQ/3.2 became stable enough for live use and by the time you're reading this, it's what you really should be using. If you are still using 2.2, here's a quick summary of the changes, and how to migrate your code.

The main change in 3.x is that PUB-SUB works properly, as in, the publisher only sends subscribers stuff they actually want. In 2.x, publishers send everything and the subscribers filter. Simple, but not ideal for performance on a TCP network.

Most of the API is backwards compatible, except a few blockheaded changes that went into 3.0 with no real regard to the cost of breaking existing code. The syntax of zmq_send[3] and zmq_recv[3] changed, and ZMQ_NOBLOCK got rebaptised to ZMQ_DONTWAIT. So although I'd love to say, *"you just recompile your code with the latest libzmq and everything will work"*, that's not how it is. For what it's worth, we banned such API breakage afterwards.

So the minimal change for C/C++ apps that use the low-level libzmq API is to replace all calls to zmq_send with zmq_msg_send, and zmq_recv with zmq_msg_recv. In other languages, your binding author may have done the work already. Note that these two functions now return -1 in case of error, and zero or more according to how many bytes were sent or received.

Other parts of the libzmq API became more consistent. We deprecated zmq_init[3] and zmq_term[3], replacing them with zmq_ctx_new[3] and zmq_ctx_destroy[3]. We added zmq_ctx_set[3] to let you configure a context before starting to work with it.

Finally, we added context monitoring via the zmq_ctx_set_monitor[3] call, which lets you track connections and disconnections, and other events on sockets.

1.17. Warning - Unstable Paradigms!

Traditional network programming is built on the general assumption that one socket talks to one connection, one peer. There are multicast protocols but these 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 doorways to fast little background communications engines 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 2. Intermediate Stuff

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 multi-part 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 and use message envelopes for publish-subscribe.
- Using the high-water mark (HWM) to protect against memory overflows.

2.1. The Zen of Zero

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 descendants 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.

2.2. The Socket API

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 socket-based API but that hides a bunch of message-processing engines 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 sockets and messages instead of some other arbitrary set of concepts. Kudos to Martin Sustrik for pulling this 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_msg_send[3], zmq_msg_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_msg_recv (&mouse, mousetrap, 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_msg_send[3] and zmq_msg_recv[3]. As a mnemonic, realize that "in ØMQ all your 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.

2.3. Plugging Sockets Into the Topology

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 static, 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 static 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.

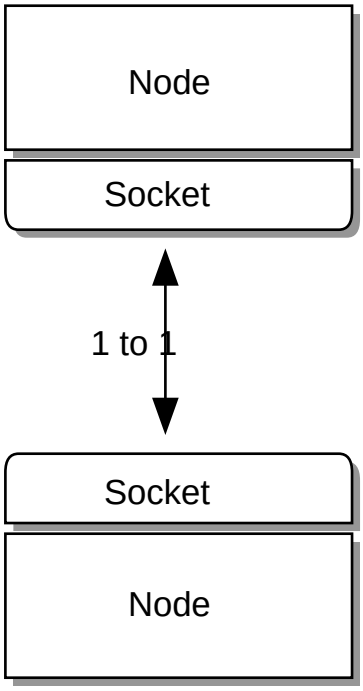
Sockets have types. The socket type defines the semantics of the socket, its policies for routing messages inwards and outwards, queuing, etc. You can connect certain types of socket together, e.g. a publisher socket and a subscriber socket. Sockets work together in "messaging patterns". We'll look at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

2.4. Using Sockets to Carry Data

To send and receive messages you use the zmq_msg_send[3] and zmq_msg_recv[3] methods. The names are conventional but ØMQ's I/O model is different enough from the TCP model(Figure 2-1) that you will need time to get your head around it.

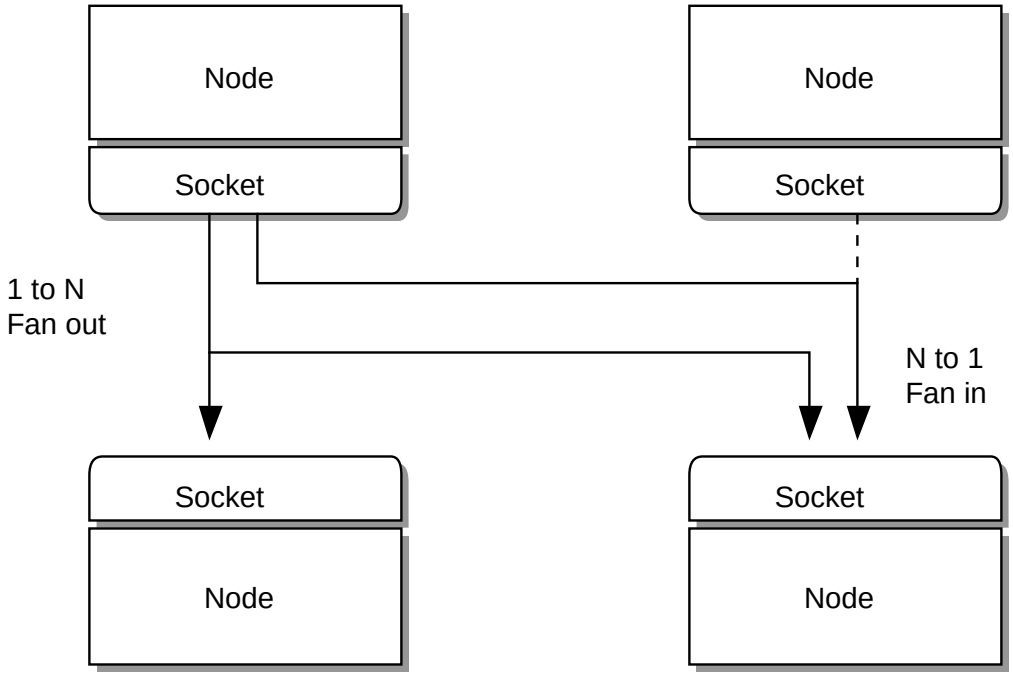
Figure 2-1. TCP sockets are 1 to 1



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 2-2).

Figure 2-2. 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_msg_recv[3] method uses a fair-queuing algorithm so each sender gets an even chance.

The zmq_msg_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_msg_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.

2.5. Unicast Transports

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.

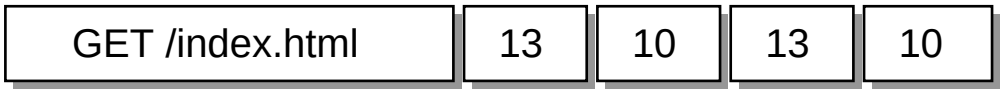
2.6. ØMQ is Not a Neutral Carrier

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.

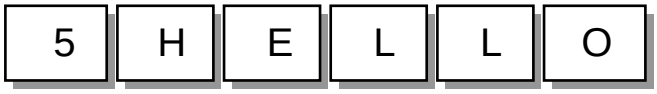
Sadly the answer is "this is not how it works". ØMQ is not a neutral carrier, it imposes a framing on the transport protocols it uses. This framing is not compatible with existing protocols, which tend to use their own framing. For example, compare an HTTP request(Figure 2-3), and a ØMQ request, both over TCP/IP.

Figure 2-3. HTTP On the Wire



Where the HTTP request uses CR-LF as its simplest framing delimiter, and ØMQ uses a length-specified frame(Figure 2-4).

Figure 2-4. ØMQ On the Wire



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 (http://www.mongrel2.org) web server is a great example of such an architecture.

2.7. I/O Threads

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. When you create a new context it starts with one I/O thread. The general rule of thumb is to allow one I/O thread per gigabyte of data in or out per second. To raise the number of I/O threads, use the zmq_ctx_set[3] call *before* creating any sockets:

```
int io_threads = 4;
void *context = zmq_ctx_new ();
zmq_ctx_set (context, ZMQ_IO_THREADS, io_threads);
assert (zmq_ctx_get (context, ZMQ_IO_THREADS) == io_threads);
```

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 outgoing 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.

2.8. Limiting Socket Use

By default, a ØMQ socket will continue to accept connections until your operating system runs out of file handles. This isn't always the best policy for public-facing services as it leaves you open to a simple denial-of-service attack. You can set a limit using another zmq_ctx_set[3] call:

```
int max_sockets = 1024;
void *context = zmq_ctx_new ();
zmq_ctx_get (context, ZMQ_MAX_SOCKETS, max_sockets);
assert (zmq_ctx_get (context, ZMQ_MAX_SOCKETS) == max_sockets);
```

2.9. Core Messaging Patterns

Underneath the brown paper wrapping of ØMQ's socket API lies the world of messaging patterns. If you have a background in enterprise messaging, or know UDP well, 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 maps one-to-one to 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 pattern you should use only to connect two threads in a process. 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. 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

You'll also see references to XPUB and XSUB sockets, which we'll come to later (they're like raw versions of PUB and SUB). 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.

2.10. High-level Messaging Patterns

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.

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. For example the Majordomo pattern, which we explore in Chapter Four, sits in the github Majordomo project in the ZeroMQ organization.

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).

2.11. Working with Messages

On the wire, ØMQ messages are blobs of any size from zero upwards, fitting in memory. You do your own serialization using protobufs, msgpack, 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_msg_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[3], and pass the message to zmq_msg_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);
    int size = zmq_msg_recv (&message, socket, 0);
    if (size == -1)
        return NULL;
    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) {
    zmq_msg_t message;
    zmq_msg_init_size (&message, strlen (string));
    memcpy (zmq_msg_data (&message), string, strlen (string));
    int size = zmq_msg_send (&message, socket, 0);
    zmq_msg_close (&message);
    return (size);
}
```

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

Note than when you have passed a message to zmq_msg_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 *multi-part* messages, which let you send or receive a list of frames as a single on-the-wire 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. This is also true for multi-part messages.
- ØMQ does not send a message right away but at some indeterminate later time.
- You may 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.

2.12. Handling Multiple Sockets

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 fan-in for us. This is legal if the remote endpoints are in the same pattern but it would be wrong to e.g. connect a PULL socket to a PUB endpoint.

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:

Example 2-1. Multiple socket reader (msreader.lua)

```
--
--  Reading from multiple sockets
--  This version uses a simple recv loop
--
--  Author: Robert G. Jakabosky <bobby@sharedrealm.com>
--
require"zmq"
require"zhelpers"

--  Prepare our context and sockets
local context = zmq.init(1)

--  Connect to task ventilator
local receiver = context:socket(zmq.PULL)
receiver:connect("tcp://localhost:5557")

--  Connect to weather server
local subscriber = context:socket(zmq.SUB)
subscriber:connect("tcp://localhost:5556")
subscriber:setopt(zmq.SUBSCRIBE, "10001 ")

--  Process messages from both sockets
--  We prioritize traffic from the task ventilator
while true do
    --  Process any waiting tasks
    local msg
```

```
    while true do
        msg = receiver:recv(zmq.NOBLOCK)
        if not msg then break end
        --  process task
    end
    --  Process any waiting weather updates
    while true do
        msg = subscriber:recv(zmq.NOBLOCK)
        if not msg then break end
        --  process weather update
    end
    --  No activity, so sleep for 1 msec
    s_sleep (1)
end
--  We never get here but clean up anyhow
receiver:close()
subscriber:close()
context:term()
```

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]:

Example 2-2. Multiple socket poller (mspoller.lua)

```
--
--  Reading from multiple sockets
--  This version uses :poll()
--
--  Author: Robert G. Jakabosky <bobby@sharedrealm.com>
--
require"zmq"
require"zmq.poller"
require"zhelpers"

local context = zmq.init(1)

--  Connect to task ventilator
local receiver = context:socket(zmq.PULL)
receiver:connect("tcp://localhost:5557")

--  Connect to weather server
local subscriber = context:socket(zmq.SUB)
```

```
subscriber:connect("tcp://localhost:5556")
subscriber:setopt(zmq.SUBSCRIBE, "10001 ", 6)

local poller = zmq.poller(2)

poller:add(receiver, zmq.POLLIN, function()
    local msg = receiver:recv()
    --  Process task
end)

poller:add(subscriber, zmq.POLLIN, function()
    local msg = subscriber:recv()
    --  Process weather update
end)

--  Process messages from both sockets
-- start poller's event loop
poller:start()

--  We never get here
receiver:close()
subscriber:close()
context:term()
```

2.13. Handling Errors and ETERM

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

When ØMQ detects an external fault 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 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 if they fail.
- Methods that process data may return the number of bytes processed, or -1 on an error or failure.

- Other methods will return 0 on success and -1 on an error or 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_msg_recv[3] with the ZMQ_DONTWAIT option and there is no waiting data. ØMQ will return -1 and set errno to EAGAIN.
- When a thread calls zmq_ctx_destroy[3] and other threads are doing blocking work. The zmq_ctx_destroy[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_ctx_new ();
assert (context);
void *socket = zmq_socket (context, ZMQ_REP);
assert (socket);
int rc = zmq_bind (socket, "tcp://*:5555");
if (rc != 0) {
    printf ("E: bind failed: %s\n", strerror (errno));
    return -1;
}
```

In C/C++, asserts can be removed entirely in optimized code, so don't make the mistake of wrapping the whole ØMQ call in an assert(). It looks neat, then the optimizer removes all the asserts and the calls you want to make, and your application breaks in impressive ways.

Let's see how to shut down a process cleanly. We'll take the parallel pipeline example from the previous section. If we've started 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(Figure 2-5):

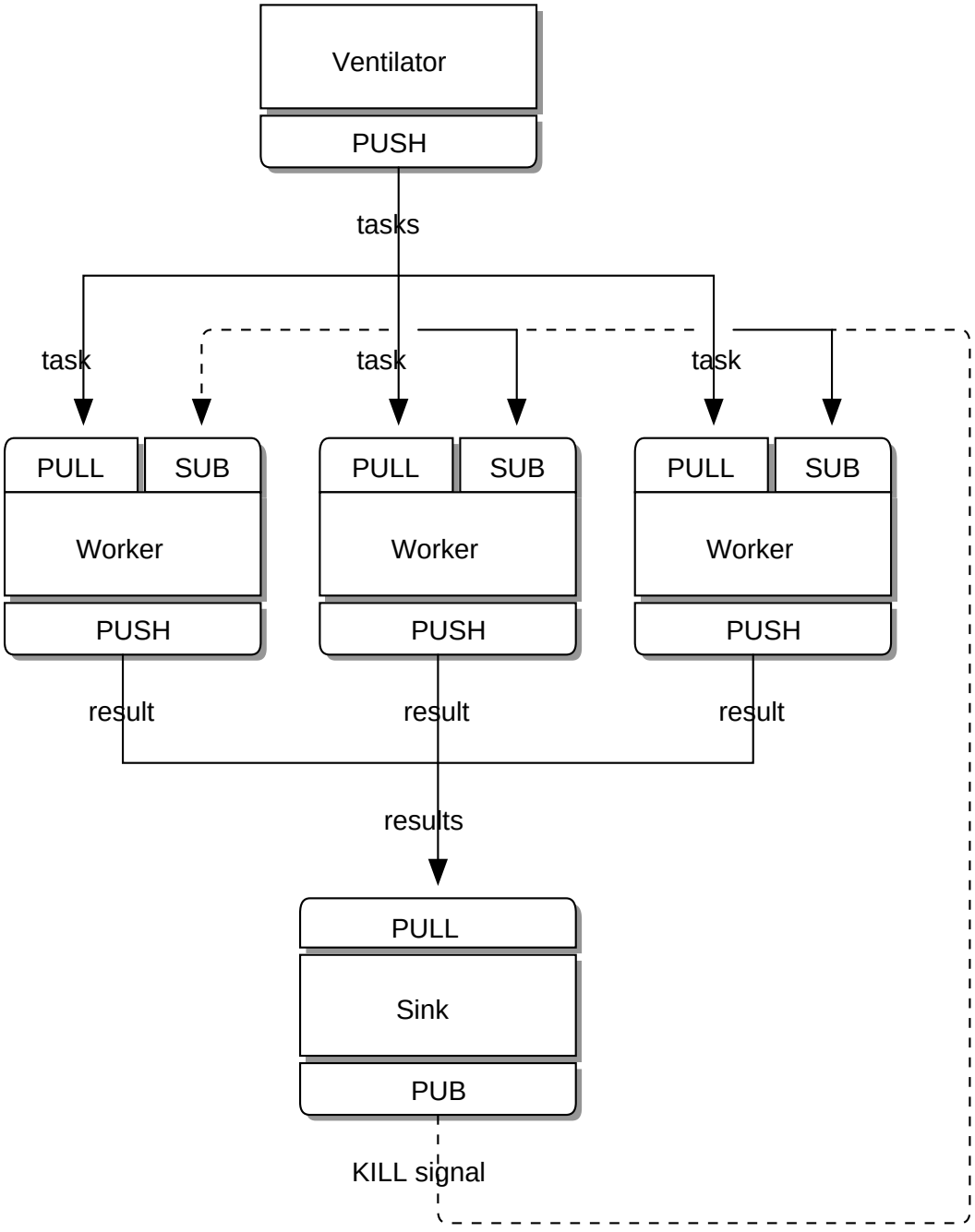
- The sink creates a PUB socket on a new endpoint.
- Workers bind their input socket to this endpoint.
- When the sink detects the end of the batch it sends a kill to its PUB socket.
- When a worker detects this kill message, it exits.

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_msg_send (control, &message, 0);
zmq_msg_close (&message);
```

Figure 2-5. 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:

Example 2-3. Parallel task worker with kill signaling (taskwork2.lua)

```
--
--  Task worker - design 2
--  Adds pub-sub flow to receive and respond to kill signal
--
--  Author: Robert G. Jakabosky <bobby@sharedrealm.com>
--
require"zmq"
require"zmq.poller"
require"zhelpers"

local context = zmq.init(1)

--  Socket to receive messages on
local receiver = context:socket(zmq.PULL)
receiver:connect("tcp://localhost:5557")

--  Socket to send messages to
local sender = context:socket(zmq.PUSH)
sender:connect("tcp://localhost:5558")

--  Socket for control input
local controller = context:socket(zmq.SUB)
controller:connect("tcp://localhost:5559")
controller:setopt(zmq.SUBSCRIBE, "", 0)

--  Process messages from receiver and controller
local poller = zmq.poller(2)
poller:add(receiver, zmq.POLLIN, function()
    local msg = receiver:recv()

    --  Do the work
    s_sleep(tonumber(msg))

    --  Send results to sink
    sender:send("")

    --  Simple progress indicator for the viewer
    io.write(".")
    io.stdout:flush()
end)
poller:add(controller, zmq.POLLIN, function()
    poller:stop() --  Exit loop
end)

-- start poller's event loop
poller:start()

--  Finished
receiver:close()
```

```
sender:close()
controller:close()
context:term()
```

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

Example 2-4. Parallel task sink with kill signaling (tasksink2.lua)

```
--
--  Task sink - design 2
--  Adds pub-sub flow to send kill signal to workers
--
--  Author: Robert G. Jakabosky <bobby@sharedrealm.com>
--
require"zmq"
require"zhelpers"
local fmod = math.fmod

local context = zmq.init(1)

--  Socket to receive messages on
local receiver = context:socket(zmq.PULL)
receiver:bind("tcp://*:5558")

--  Socket for worker control
local controller = context:socket(zmq.PUB)
controller:bind("tcp://*:5559")

--  Wait for start of batch
local msg = receiver:recv()

--  Start our clock now
local start_time = s_clock ()

--  Process 100 confirmations
local task_nbr
for task_nbr=0,99 do
    local msg = receiver:recv()

    if (fmod(task_nbr, 10) == 0) then
        printf (":")
    else
        printf (".")
    end
    io.stdout:flush()
end
printf("Total elapsed time: %d msec\n", (s_clock () - start_time))

--  Send kill signal to workers
controller:send("KILL")
```

```
--  Finished
s_sleep (1000)              --  Give 0MQ time to deliver

receiver:close()
controller:close()
context:term()
```

2.14. Handling Interrupt Signals

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:

Example 2-5. Handling Ctrl-C cleanly (interrupt.lua)

```
--
--  Shows how to handle Ctrl-C
--
--  Author: Robert G. Jakabosky <bobby@sharedrealm.com>
--
require"zmq"
require"zhelpers"

local context = zmq.init(1)
local server = context:socket(zmq.REP)
server:bind("tcp://*:5555")

s_catch_signals ()
while true do
    --  Blocking read will exit on a signal
    local request = server:recv()
    if (s_interrupted) then
        printf ("W: interrupt received, killing server...\n")
        break
    end
    server:send("World")
end
server:close()
context:term()
```

The program provides s_catch_signals(), which traps Ctrl-C (SIGINT) and SIGTERM. When either of these signals arrive, the s_catch_signals() handler sets the global variable s_interrupted. Your application will not die automatically, you have to now explicitly check for an interrupt, and handle it properly. Here's how:

- Call s_catch_signals() (copy this from interrupt.c) at the start of your main code. This sets-up the signal handling.
- If your code is blocking in zmq_msg_recv[3], zmq_poll[3], or zmq_msg_send[3], when a signal arrives, the call will return with EINTR.
- Wrappers like s_recv() return NULL if they are interrupted.
- So, your application checks for an EINTR return code, a NULL return, and/or s_interrupted.

Here is a typical code fragment:

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

If you call s_catch_signals() and don't test for interrupts, the your application will become immune to Ctrl-C and SIGTERM, which may be useful, but is usually not.

2.15. Detecting Memory Leaks

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, create a file `valgrind.supp` that contains this:

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

- 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, 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 --suppressions=valgrind.supp someprog
```

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

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

2.16. Multi-part Messages

ØMQ lets us compose a message out of several frames, giving us a 'multi-part message'. Realistic applications use multi-part messages heavily, both for wrapping messages with address information, and for simple serialization. We'll look at address envelopes later.

What we'll learn now is simply how to safely (but blindly) read and write multi-part messages in any application (like a proxy) that needs to forward messages without inspecting them.

When you work with multi-part 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 multi-part message (we receive each frame into a message object):

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

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

```
while (1) {
    zmq_msg_t message;
    zmq_msg_init (&message);
    zmq_msg_recv (socket, &message, 0);
    //  Process the message frame
    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 frame
}
```

Some things to know about multi-part messages:

- When you send a multi-part message, the first part (and all following parts) are only actually sent on the wire 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 has 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 frames 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.

2.17. Intermediaries and Proxies

ØMQ aims for decentralized intelligence but that doesn't mean your network is empty space in the middle. It's filled with message-aware infrastructure and quite often, we build that infrastructure with ØMQ. The ØMQ plumbing can range from tiny pipes to full-blown service-oriented brokers. The messaging industry calls this "intermediation", meaning that the stuff in the middle deals with either side. In ØMQ we call these proxies, queues, forwarders, device, or brokers, depending on the context.

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. Real-world intermediaries are typically called wholesalers, distributors, managers, etc.

2.17.1. The Dynamic Discovery Problem

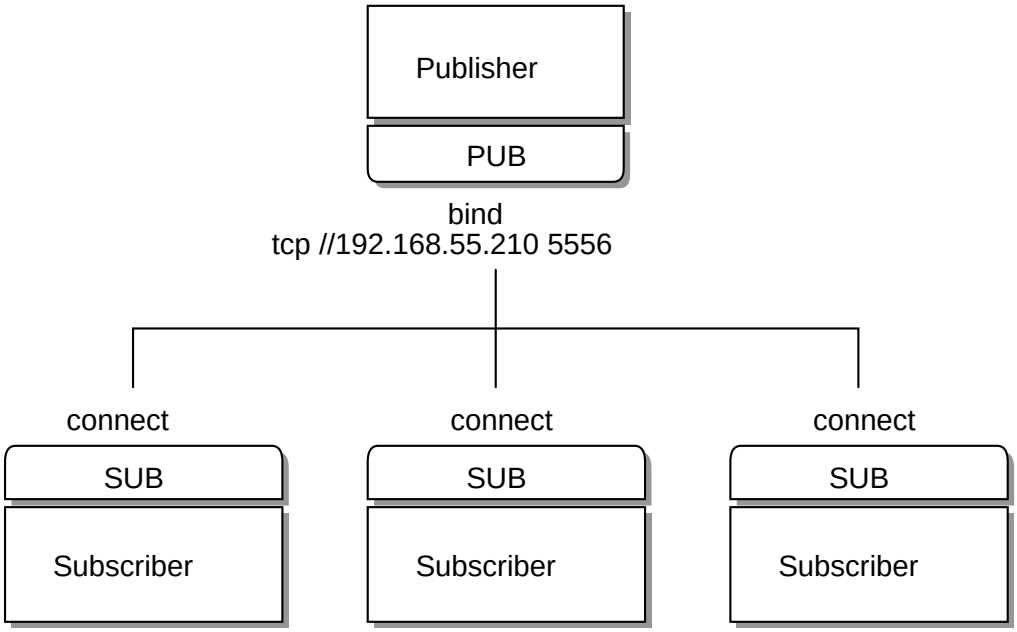
One of the problems you will hit as you design larger distributed architectures is discovery. That is, how do pieces know about each other? It's especially difficult if pieces come and go, thus we can call this the "dynamic discovery problem".

There are several solutions to dynamic discovery. The simplest is to entirely avoid it by hard-coding (or configuring) the network architecture so discovery is done by hand. That is, when you add a new piece, you reconfigure the network to know about it.

In practice this leads to increasingly fragile and hard-to-manage architectures. Let's say you have one publisher and a hundred subscribers. You connect each subscriber to the publisher by configuring a

publisher endpoint in each subscriber. That's easy(Figure 2-6). Subscribers are dynamic, the publisher is static. Now say you add more publishers. Suddenly it's not so easy any more. If you continue to connect each subscriber to each publisher, the cost of avoiding dynamic discovery gets higher and higher.

Figure 2-6. Small-scale Pub-Sub Network

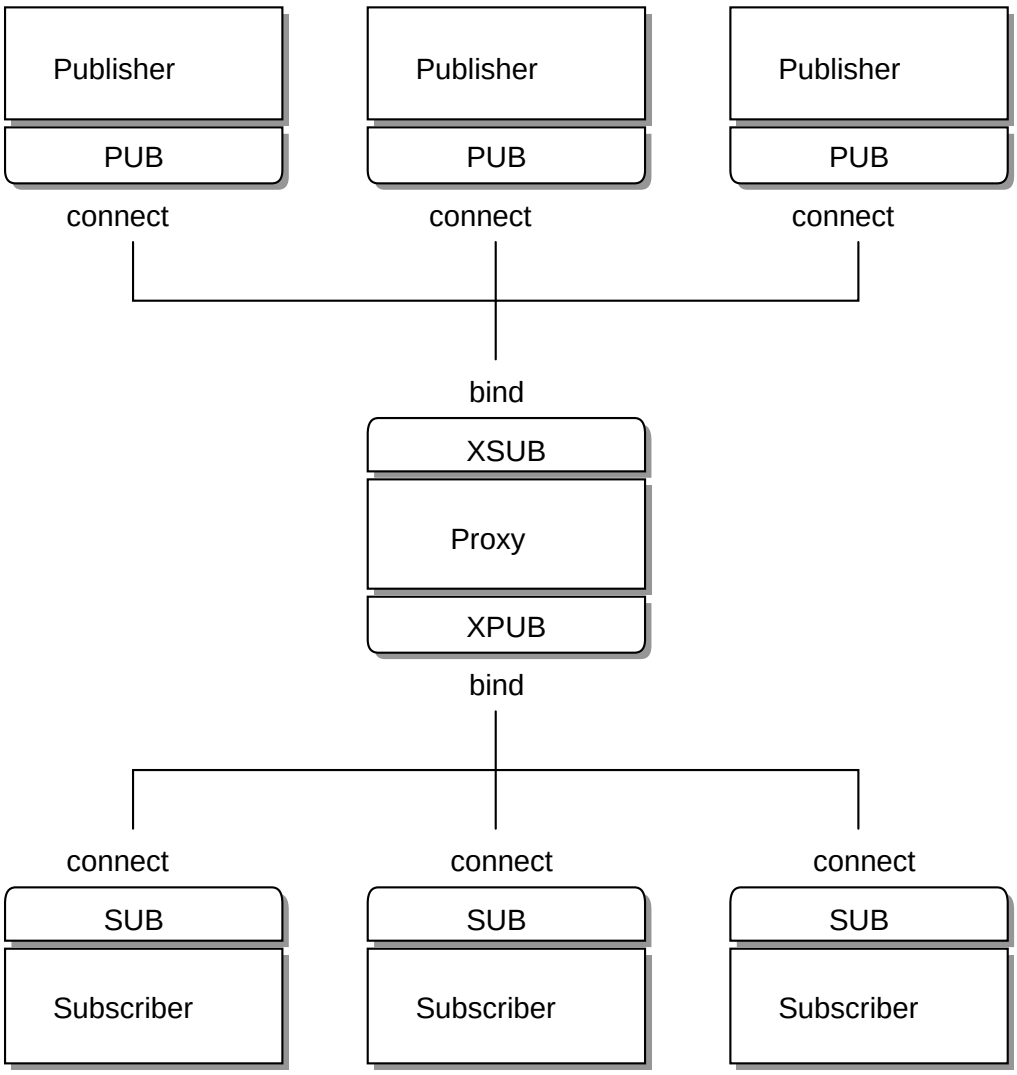


There are quite a few answers to this but the very simplest answer is to add an intermediary, that is, a static point in the network to which all other nodes connect. In classic messaging, this is the job of the "message broker". ØMQ doesn't come with a message broker as such, but it lets us build intermediaries quite easily.

You might wonder, if all networks eventually get large enough to need intermediaries, why don't we simply always design around a message broker? For beginners, it's a fair compromise. Just always use a star topology, forget about performance, and things will usually work. However message brokers are greedy things; in their role as central intermediaries, they become too complex, too stateful, and eventually a problem.

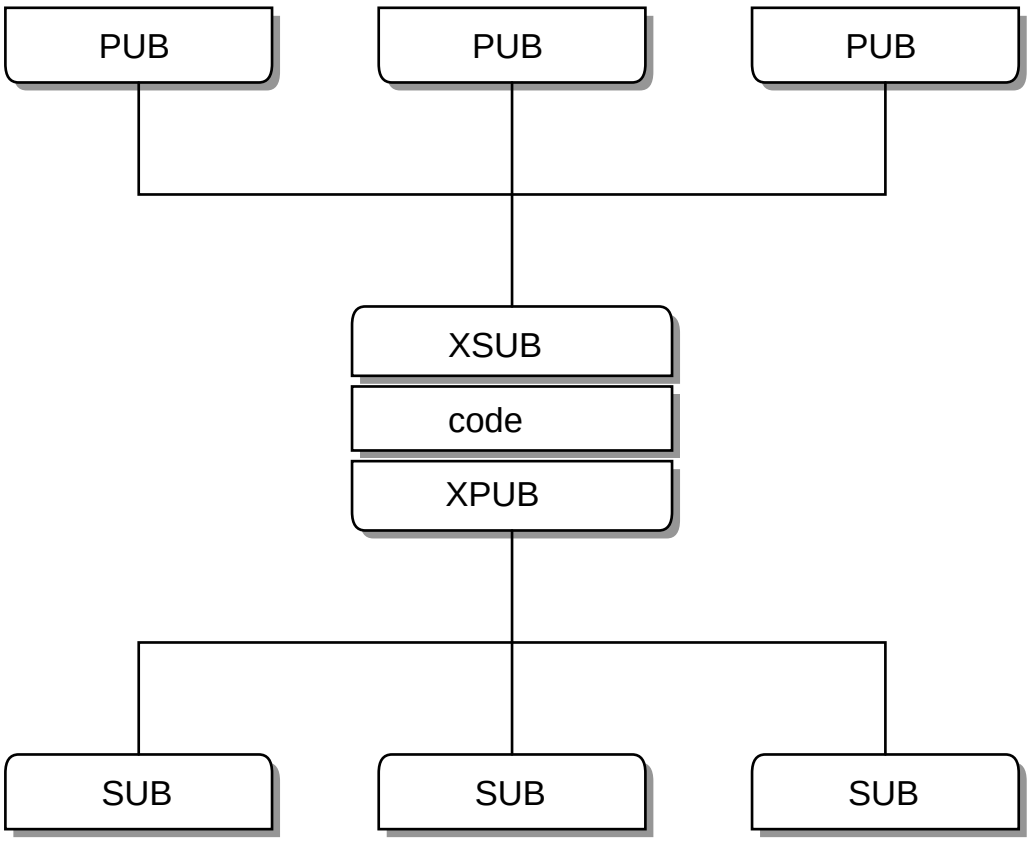
It's better to think of intermediaries as simple stateless message switches. The best analogy is an HTTP proxy; it's there but doesn't have any special role. Adding a pub-sub proxy solves the dynamic discovery problem in our example. We set the proxy in the "middle" of the network(Figure 2-7). The proxy opens an XSUB socket, an XPUB socket, and binds each to well-known IP addresses and ports. Then all other processes connect to the proxy, instead of to each other. It becomes trivial to add more subscribers or publishers.

Figure 2-7. Pub-Sub Network with a Proxy



We need XPUB and XSUB sockets because ØMQ does subscription forwarding: SUB sockets actually send subscriptions to PUB sockets as special messages. The proxy has to forward these as well, by reading them from the XPUB socket and writing them to the XSUB socket. This is the main use-case for XSUB and XPUB(Figure 2-8).

Figure 2-8. Extended Publish-Subscribe

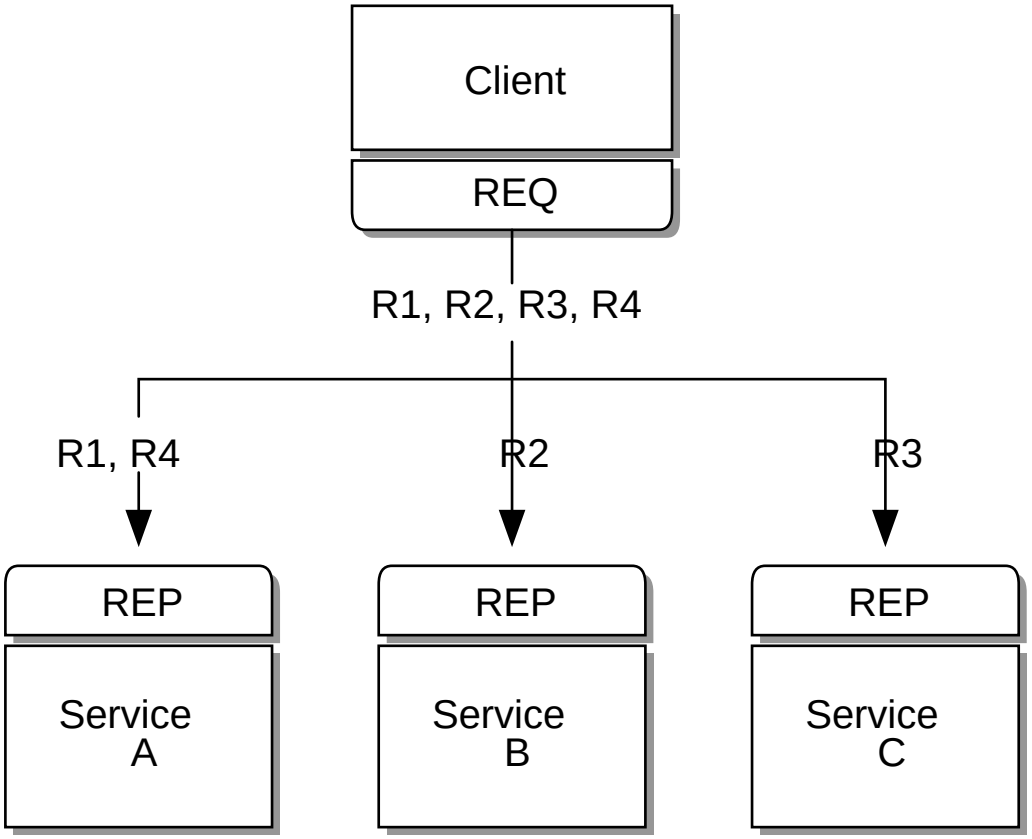


2.17.2. The Shared Queue Problem

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 the REQ socket will then load-balance requests 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 2-9).

Figure 2-9. 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 static pieces are like liquid concrete: knowledge is distributed and the more static 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.

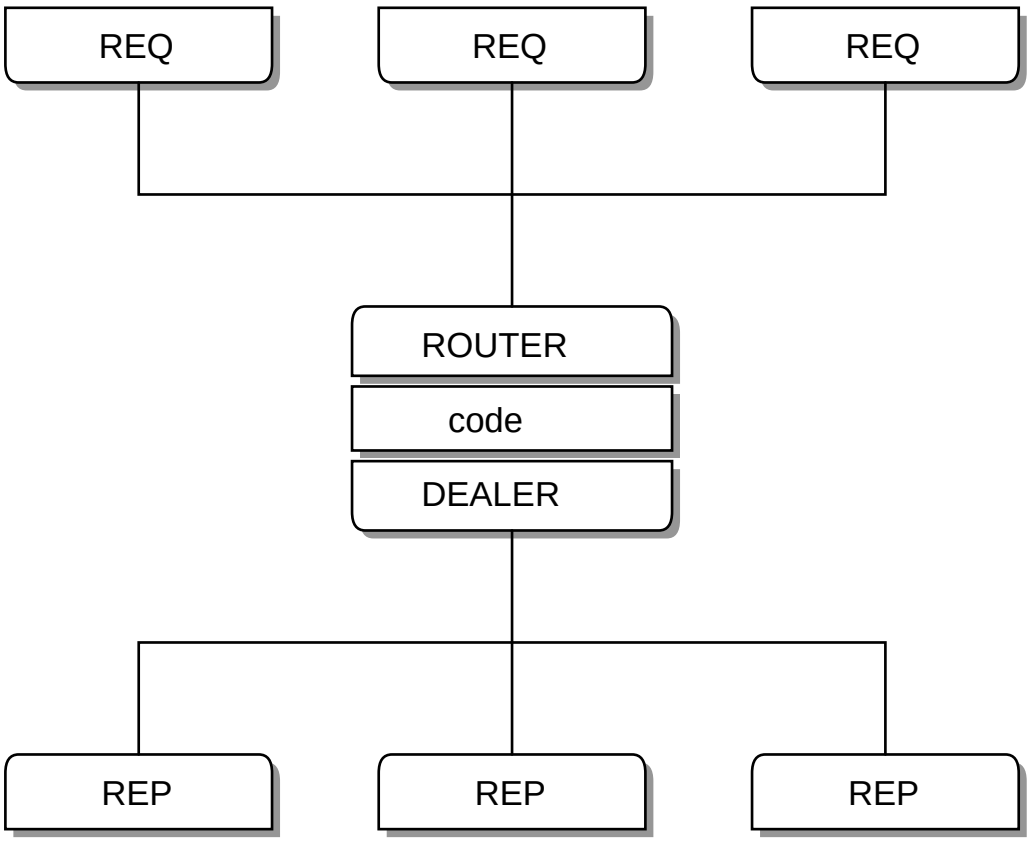
When you use REQ to talk to REP you get a strictly synchronous request-reply dialog. The client sends a request, the service reads the request and sends a reply. The client then reads the reply. If either the client or the service try to do anything else (e.g. sending two requests in a row without waiting 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. You'll see in Chapter Three how DEALER and ROUTER sockets let you build all kinds of asynchronous request-reply flows. For now, we're just going to see how DEALER and ROUTER let us extend REQ-REP across an intermediary, 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 2-10).

Figure 2-10. Extended Request-reply



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

Example 2-6. Request-reply client (rrclient.lua)

```
--
--  Hello World client
--  Connects REQ socket to tcp://localhost:5559
--  Sends "Hello" to server, expects "World" back
--
--  Author: Robert G. Jakabosky <bobby@sharedrealm.com>
--
require"zmq"
require"zhelpers"
```

```
local context = zmq.init(1)

--  Socket to talk to server
local requester = context:socket(zmq.REQ)
requester:connect("tcp://localhost:5559")

for n=0,9 do
    requester:send("Hello")
    local msg = requester:recv()
    printf ("Received reply %d [%s]\n", n, msg)
end
requester:close()
context:term()
```

Here is the worker:

Example 2-7. Request-reply worker (rrworker.lua)

```
--
--  Hello World server
--  Connects REP socket to tcp://*:5560
--  Expects "Hello" from client, replies with "World"
--
--  Author: Robert G. Jakabosky <bobby@sharedrealm.com>
--
require"zmq"
require"zhelpers"

local context = zmq.init(1)

--  Socket to talk to clients
local responder = context:socket(zmq.REP)
responder:connect("tcp://localhost:5560")

while true do
    --  Wait for next request from client
    local msg = responder:recv()
    printf ("Received request: [%s]\n", msg)

    --  Do some 'work'
    s_sleep (1000)

    --  Send reply back to client
    responder:send("World")
end
--  We never get here but clean up anyhow
responder:close()
context:term()
```

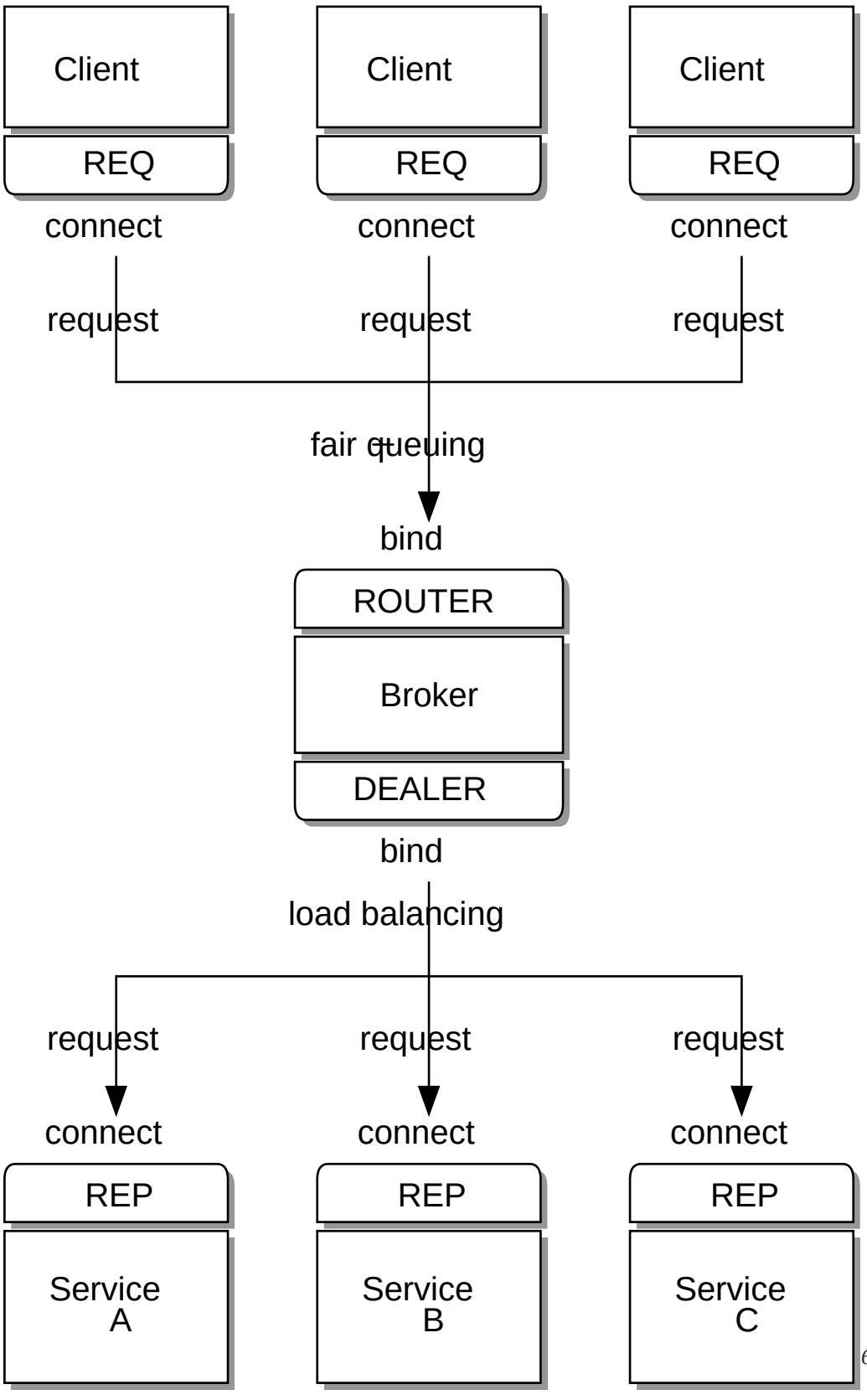
And here is the broker, which properly handles multi-part messages:

Example 2-8. Request-reply broker (rrbroker.lua)

```
--
--  Simple request-reply broker
--
--  Author: Robert G. Jakabosky <bobby@sharedrealm.com>
--
require"zmq"
require"zmq.poller"
require"zhelpers"

--  Prepare our context and sockets
local context = zmq.init(1)
local frontend = context:socket(zmq.ROUTER)
local backend  = context:socket(zmq.DEALER)
frontend:bind("tcp://*:5559")
backend:bind("tcp://*:5560")

--  Switch messages between sockets
local poller = zmq.poller(2)
poller:add(frontend, zmq.POLLIN, function()
    while true do
        --  Process all parts of the message
        local msg = frontend:recv()
        if (frontend:getopt(zmq.RCVMORE) == 1) then
            backend:send(msg, zmq.SNDMORE)
        else
            backend:send(msg, 0)
            break;      --  Last message part
        end
    end
end)
poller:add(backend, zmq.POLLIN, function()
    while true do
        --  Process all parts of the message
        local msg = backend:recv()
        if (backend:getopt(zmq.RCVMORE) == 1) then
            frontend:send(msg, zmq.SNDMORE)
        else
            frontend:send(msg, 0)
            break;      --  Last message part
        end
    end
end)

-- start poller's event loop
poller:start()

--  We never get here but clean up anyhow
frontend:close()
backend:close()
context:term()
```

Using a request-reply broker makes your client-server architectures easier to scale since clients don't see workers, and workers don't see clients. The only static node is the broker in the middle(Figure 2-11).

Figure 2-11. Request-reply Broker



2.17.3. ØMQ's Built-in Proxy Function

It turns out that that core loop in rrbroker is very useful, and reusable. It lets us build pub-sub forwarders and shared queues and other little intermediaries, with very little effort. ØMQ wraps this up in a single method, zmq_proxy[3]:

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

The two (or three sockets, if we want to capture data) must be properly connected, bound, configured. When we call the zmq_proxy method it's exactly like starting the main loop of rrbroker. Let's rewrite the request-reply broker to call zmq_proxy, and re-badge this as an expensive-sounding "message queue" (people have charged houses for code that did less):

Example 2-9. Message queue broker (msgqueue.lua)

```
--
--  Simple message queuing broker
--  Same as request-reply broker but using QUEUE device
--
--  Author: Robert G. Jakabosky <bobby@sharedrealm.com>
--
require"zmq"
require"zhelpers"

local context = zmq.init(1)

--  Socket facing clients
local frontend = context:socket(zmq.ROUTER)
frontend:bind("tcp://*:5559")

--  Socket facing services
local backend = context:socket(zmq.DEALER)
backend:bind("tcp://*:5560")

--  Start built-in device
zmq.device(zmq.QUEUE, frontend, backend)

--  We never get here...
frontend:close()
backend:close()
context:term()
```

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 the proxy?*" The short answer is: try it and work out what is happening. In practice you would usually stick to ROUTER/DEALER, XSUB/XPUB, or PULL/PUSH.

2.17.4. The Transport Bridging Problem

A frequent request from ØMQ users is "how do I connect my ØMQ network with technology X?" where X is some other networking or messaging technology. The simple answer is to build a "bridge". A bridge is a small application that speaks one protocol at one socket, and converts to/from a second protocol at another socket. A protocol interpreter, if you like. A common bridging problem in ØMQ is to bridge two transports or networks.

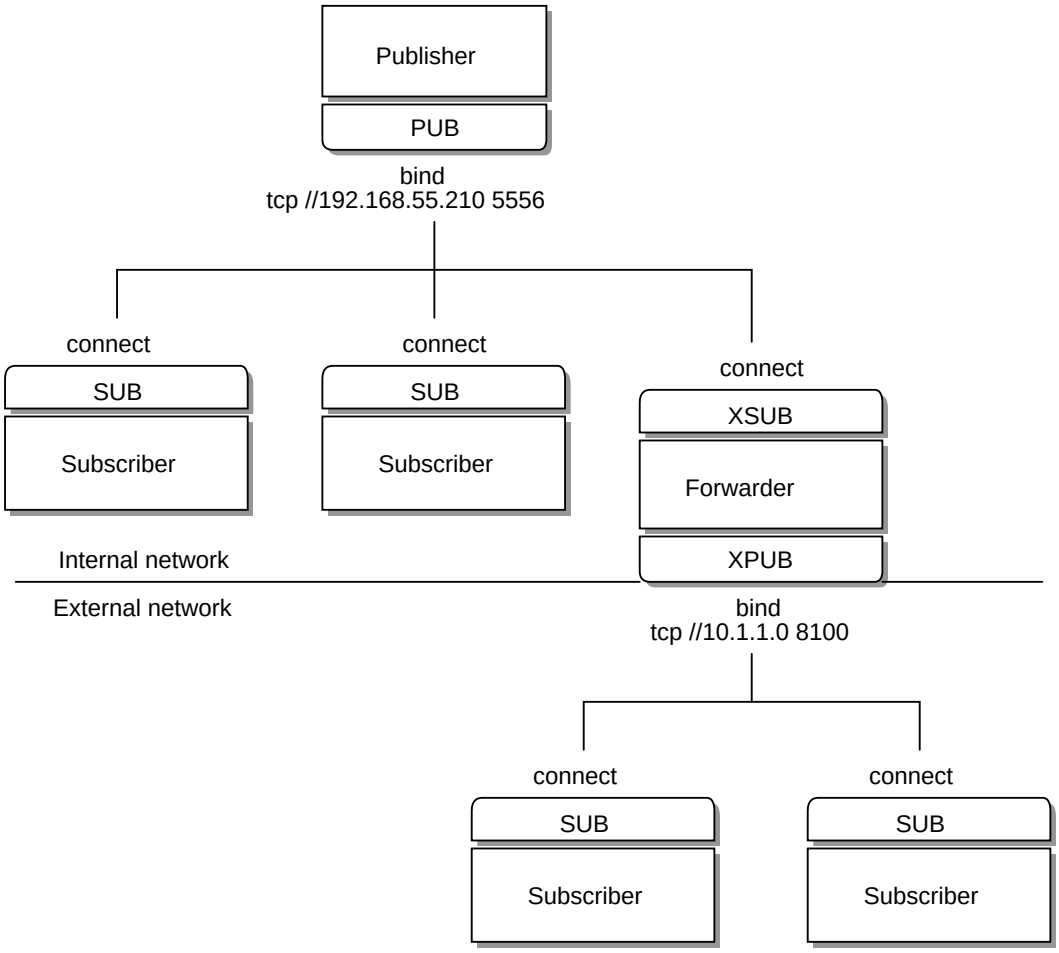
As example, we're going to write a little proxy that sits in between a publisher and a set of subscribers, bridging two networks. The frontend socket (SUB) faces the internal network, where the weather server is sitting, and the backend (PUB) faces subscribers on the external network. It subscribes to the weather service on the frontend socket, and republishes its data on the backend socket(Figure 2-12).

Example 2-10. Weather update proxy (wuproxy.lua)

```
--
--  Weather proxy device
--
--  Author: Robert G. Jakabosky <bobby@sharedrealm.com>
--
require"zmq"

local context = zmq.init(1)

--  This is where the weather server sits
local frontend = context:socket(zmq.SUB)
frontend:connect(arg[1] or "tcp://192.168.55.210:5556")

--  This is our public endpolocal for subscribers
local backend = context:socket(zmq.PUB)
backend:bind(arg[2] or "tcp://10.1.1.0:8100")

--  Subscribe on everything
frontend:setopt(zmq.SUBSCRIBE, "")

--  Shunt messages out to our own subscribers
while true do
    while true do
        --  Process all parts of the message
        local message = frontend:recv()
        if frontend:getopt(zmq.RCVMORE) == 1 then
            backend:send(message, zmq.SNDMORE)
        else
            backend:send(message)
            break      --  Last message part
        end
    end
end
--  We don't actually get here but if we did, we'd shut down neatly
frontend:close()
backend:close()
```

```
context:term()
```

Figure 2-12. Pub-Sub Forwarder Proxy



2.18. Multithreading with ØMQ

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 alight. 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 design language in any programming 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 table, 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 hilarious 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 models, despite being the basis for entire 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 models.

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 proxy 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 proxy frontend and backend sockets in one thread, and then passing the sockets to the proxy 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 start 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 proxy broker 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 broker and workers into a single process:

Example 2-11. Multithreaded service (mtserver.lua)

```
--
--  Multithreaded Hello World server
--
--  Author: Robert G. Jakabosky <bobby@sharedrealm.com>
--
require"zmq"
require"zmq.threads"
require"zhelpers"

local worker_code = [[
    local id = ...

    local zmq = require"zmq"
    require"zhelpers"
    local threads = require"zmq.threads"
    local context = threads.get_parent_ctx()
```

```
    --  Socket to talk to dispatcher
    local receiver = context:socket(zmq.REP)
    assert(receiver:connect("inproc://workers"))

    while true do
        local msg = receiver:recv()
        printf ("Received request: [%s]\n", msg)

        --  Do some 'work'
        s_sleep (1000)

        --  Send reply back to client
        receiver:send("World")
    end
    receiver:close()
    return nil
]]

s_version_assert (2, 1)
local context = zmq.init(1)

--  Socket to talk to clients
local clients = context:socket(zmq.ROUTER)
clients:bind("tcp://*:5555")

--  Socket to talk to workers
local workers = context:socket(zmq.DEALER)
workers:bind("inproc://workers")

--  Launch pool of worker threads
local worker_pool = {}
for n=1,5 do
    worker_pool[n] = zmq.threads.runstring(context, worker_code, n)
    worker_pool[n]:start()
end
--  Connect work threads to client threads via a queue
print("start queue device.")
zmq.device(zmq.QUEUE, clients, workers)

--  We never get here but clean up anyhow
clients:close()
workers:close()
context:term()
```

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

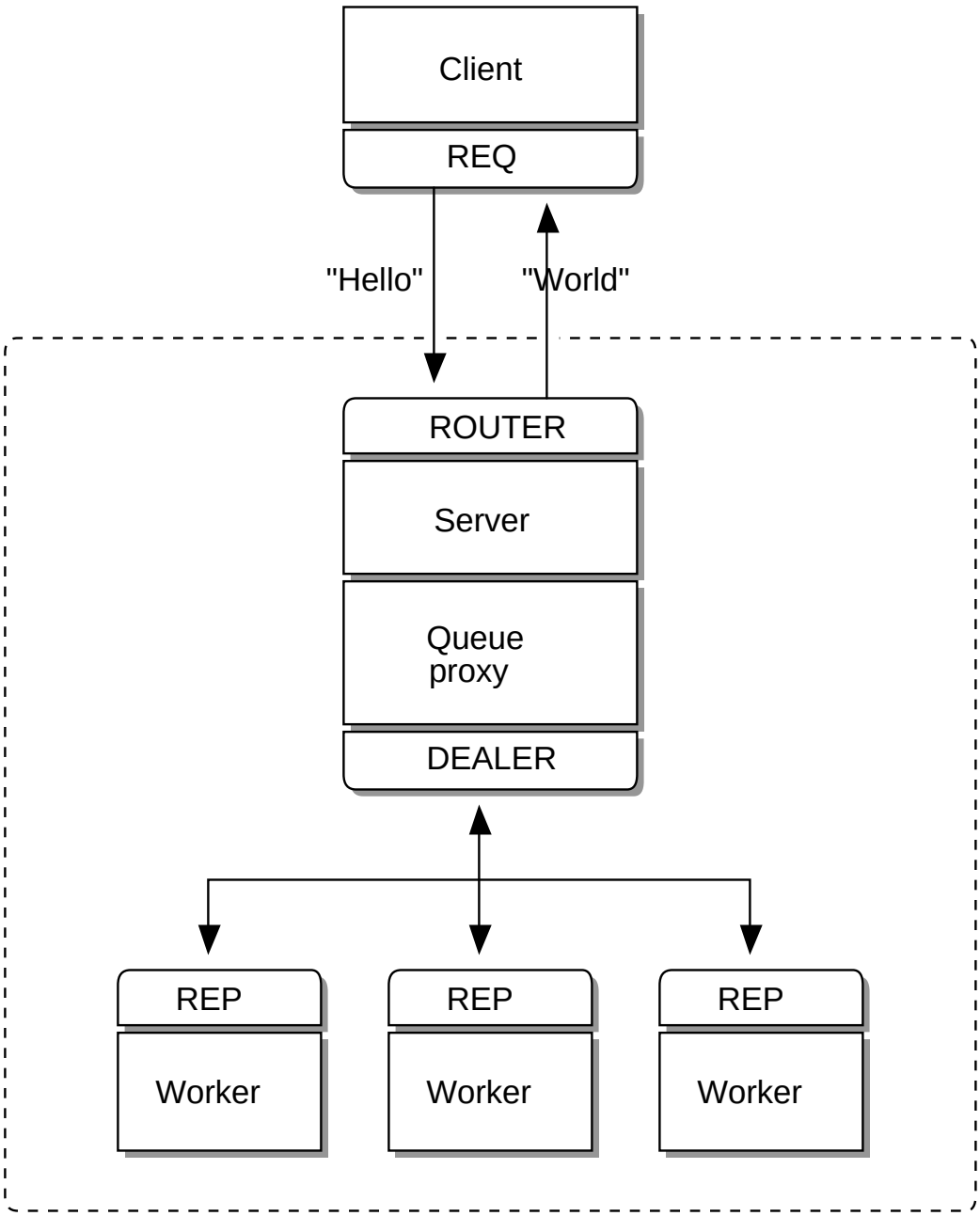
- The server starts a set of worker threads. Each worker thread creates a REP socket and then processes requests on this socket. Worker threads are just like single-threaded servers. The only differences are the transport (`inproc` instead of `tcp`), and the bind-connect direction.
- The server creates 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 proxy that connects the two sockets. The proxy pulls incoming requests fairly from all clients, 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 2-13).

Figure 2-13. Multithreaded Server



2.19. Signaling between Threads

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.

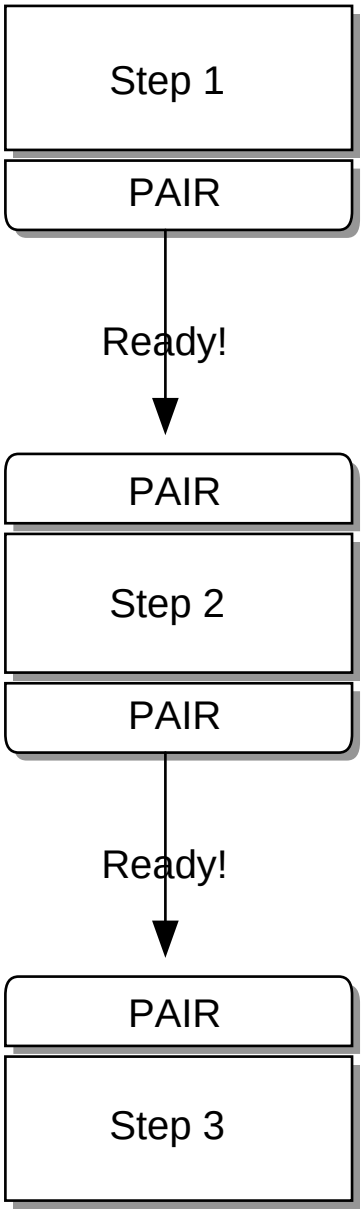
Let's make three threads that signal each other when they are ready(Figure 2-14). In this example we use PAIR sockets over the `inproc` transport:

Example 2-12. Multithreaded relay (mtrelay.lua)

```
--
--  Multithreaded relay
--
--  Author: Robert G. Jakabosky <bobby@sharedrealm.com>
--
require"zmq"
require"zhelpers"
require"zmq.threads"

local pre_code = [[
    local zmq = require"zmq"
    require"zhelpers"
    local threads = require"zmq.threads"
    local context = threads.get_parent_ctx()
]]

local step1 = pre_code .. [[
    --  Connect to step2 and tell it we're ready
    local xmitter = context:socket(zmq.PAIR)
    xmitter:connect("inproc://step2")
    xmitter:send("READY")
    xmitter:close()
]]

local step2 = pre_code .. [[
    local step1 = ...
    --  Bind inproc socket before starting step1
    local receiver = context:socket(zmq.PAIR)
    receiver:bind("inproc://step2")
    local thread = zmq.threads.runstring(context, step1)
    thread:start()

    --  Wait for signal and pass it on
    local msg = receiver:recv()

    receiver:close()

    --  Connect to step3 and tell it we're ready
    local xmitter = context:socket(zmq.PAIR)
```

```
    xmitter:connect("inproc://step3")
    xmitter:send("READY")
    xmitter:close()

    assert(thread:join())
]]

s_version_assert (2, 1)
local context = zmq.init(1)

--  Bind inproc socket before starting step2
local receiver = context:socket(zmq.PAIR)
receiver:bind("inproc://step3")
local thread = zmq.threads.runstring(context, step2, step1)
thread:start()

--  Wait for signal
local msg = receiver:recv()

receiver:close()

printf ("Test successful!\n")

assert(thread:join())

context:term()
```

Figure 2-14. The Relay Race



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 multi-part 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 your 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 messages.

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

2.20. Node Coordination

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 static. 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.

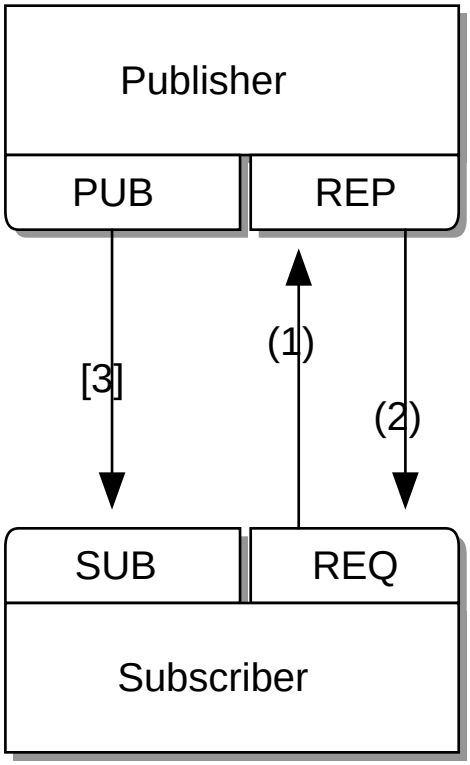
In this case we'll use a REQ-REP socket flow to synchronize subscribers and publisher(Figure 2-15). Here is the publisher:

Example 2-13. Synchronized publisher (syncpub.lua)

```
--
--  Synchronized publisher
--
--  Author: Robert G. Jakabosky <bobby@sharedrealm.com>
--
require"zmq"
require"zhelpers"

--  We wait for 10 subscribers
SUBSCRIBERS_EXPECTED   = 10

s_version_assert (2, 1)
local context = zmq.init(1)

--  Socket to talk to clients
local publisher = context:socket(zmq.PUB)
publisher:bind("tcp://*:5561")

--  Socket to receive signals
local syncservice = context:socket(zmq.REP)
syncservice:bind("tcp://*:5562")

--  Get synchronization from subscribers
local subscribers = 0
while (subscribers < SUBSCRIBERS_EXPECTED) do
    --  - wait for synchronization request
    local msg = syncservice:recv()

    --  - send synchronization reply
    syncservice:send("")
    subscribers = subscribers + 1
end
--  Now broadcast exactly 1M updates followed by END
local update_nbr
for update_nbr=1,1000000 do
    publisher:send("Rhubarb")
end

publisher:send("END")
```

```
publisher:close()
syncservice:close()
context:term()
```

Figure 2-15. Pub-Sub Synchronization



And here is the subscriber:

Example 2-14. Synchronized subscriber (syncsub.lua)

```
--
--  Synchronized subscriber
--
--  Author: Robert G. Jakabosky <bobby@sharedrealm.com>
--
require"zmq"
require"zhelpers"

local context = zmq.init(1)
```

```
--  First, connect our subscriber socket
local subscriber = context:socket(zmq.SUB)
subscriber:connect("tcp://localhost:5561")
subscriber:setopt(zmq.SUBSCRIBE, "")

--  0MQ is so fast, we need to wait a while...
s_sleep (1000)

--  Second, synchronize with publisher
local syncclient = context:socket(zmq.REQ)
syncclient:connect("tcp://localhost:5562")

--  - send a synchronization request
syncclient:send("")

--  - wait for synchronization reply
local msg = syncclient:recv()

--  Third, get our updates and report how many we got
local update_nbr = 0
while true do
    local msg = subscriber:recv()
    if (msg == "END") then
        break
    end
    update_nbr = update_nbr + 1
end
printf ("Received %d updates\n", update_nbr)

subscriber:close()
syncclient:close()
context:term()
```

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.

2.21. Zero Copy

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. To say this 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_msg_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_msg_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 multi-part 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 frames. 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.

2.22. Pub-Sub Message Envelopes

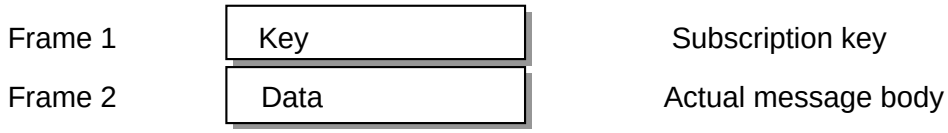
We've looked briefly at multi-part messages. Let's now look at their main use-case, which is *message envelopes*. An envelope is a way of safely packaging up data with an address, without touching the data itself.

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 2-16. 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:

Example 2-15. Pub-Sub envelope publisher (psenvpub.lua)

```
--
--  Pubsub envelope publisher
--  Note that the zhelpers.h file also provides s_sendmore
--
--  Author: Robert G. Jakabosky <bobby@sharedrealm.com>
--
require"zmq"
require"zhelpers"

--  Prepare our context and publisher
local context = zmq.init(1)
local publisher = context:socket(zmq.PUB)
publisher:bind("tcp://*:5563")

while true do
    --  Write two messages, each with an envelope and content
    publisher:send("A", zmq.SNDMORE)
    publisher:send("We don't want to see this")
    publisher:send("B", zmq.SNDMORE)
    publisher:send("We would like to see this")
    s_sleep (1000)
end
--  We never get here but clean up anyhow
publisher:close()
context:term()
```

The subscriber only wants messages of type B:

Example 2-16. Pub-Sub envelope subscriber (psenvsub.lua)

```
--
--  Pubsub envelope subscriber
--
--  Author: Robert G. Jakabosky <bobby@sharedrealm.com>
--
require"zmq"
require"zhelpers"

--  Prepare our context and subscriber
local context = zmq.init(1)
local subscriber = context:socket(zmq.SUB)
subscriber:connect("tcp://localhost:5563")
subscriber:setopt(zmq.SUBSCRIBE, "B")

while true do
    --  Read envelope with address
    local address = subscriber:recv()
    --  Read message contents
    local contents = subscriber:recv()
    printf("[%s] %s\n", address, contents)
```

```
end
--  We never get here but clean up anyhow
subscriber:close()
context:term()
```

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 multi-part message (key plus data). You won't get part of a multi-part 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 2-17. Pub-Sub Envelope with Sender Address

2.23. High Water Marks

When you can send messages rapidly from process to process, you soon discover that memory is a precious resource, and one that's trivially filled up. A few seconds delay somewhere in a process can turn into a backlog that blows up a server, unless you understand the problem and take precautions.

The problem is this: if you have process A sending messages to process B, which suddenly gets very busy (garbage collection, CPU overload, whatever), then what happens to the messages that process A wants to send? Some will sit in B's network buffers. Some will sit on the Ethernet wire itself. Some will sit in A's network buffers. And the rest will accumulate in A's memory. If you don't take some precaution, A can easily run out of memory and crash. It is a consistent, classic problem with message brokers.

What are the answers? One is to pass the problem upstream. A is getting the messages from somewhere else. So tell that process, "stop!" And so on. This is called "flow control". It sounds great, but what if you're sending out a Twitter feed? Do you tell the whole world to stop tweeting while B gets its act together?

Flow control works in some cases but in others, the transport layer can't tell the application layer "stop" any more than a subway system can tell a large business, "please keep your staff at work another half an hour, I'm too busy".

The answer for messaging is to set limits on the size of buffers, and then when we reach those limits, take some sensible action. In most cases (not for a subway system, though), the answer is to throw away messages. In a few others, it's to wait.

ØMQ uses the concept of "high water mark" or HWM to define the capacity of its internal pipes. Each connection out of a socket or into a socket has its own pipe, and HWM capacity.

In ØMQ/2.x the HWM was set to infinite by default. In ØMQ/3.x it's set to 1,000 by default, which is more sensible. If you're still using ØMQ/2.x you should always set a HWM on your sockets, be it 1,000 to match ØMQ/3.x or another figure that takes into account your message sizes.

The high water mark 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 receiver 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.

2.24. A Bare Necessity

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 (http://en.wikipedia.org/wiki/Flow-based_programming) and languages like Erlang (http://www.erlang.org/) already worked like this - but ØMQ makes it easier to use than ever before.

As Gonzo Diethelm said (http://permalink.gmane.org/gmane.network.zeromq.devel/2145), *'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 3. 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.

3.1. 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 multi-part data, so if you copy multi-part 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 the ROUTER socket 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 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, 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 (e.g. by sending to a DEALER connected to a ROUTER), the second ROUTER will in turn stick another brown envelope on it, and scribble the name of that DEALER on it.

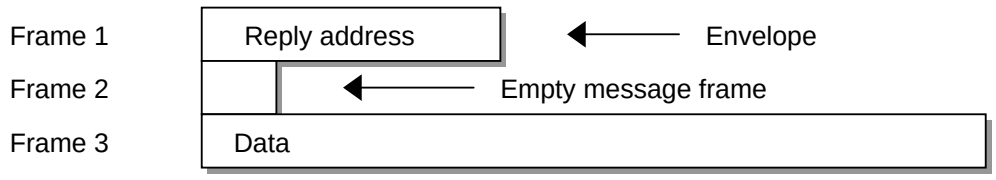
The whole point of this is that each ROUTER knows how to send replies back to the right place. All you need to do, in your application, is respect the brown envelopes. Now the REP socket makes sense. It carefully slices open the brown envelopes, one by one, keeps them safely aside, and gives you (the 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 ROUTERs 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, and send one request message, you will get a message that consists of three frames: a reply address, an empty message frame, and the 'real' message(Figure 3-1).

Figure 3-1. Single-hop Request-reply Envelope

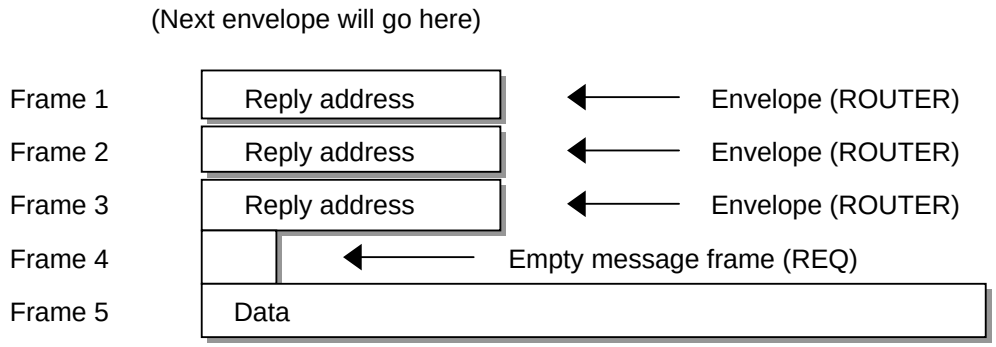


Breaking this down:

- The data in frame 3 is what the sending application sends to the REQ socket.
- The empty message frame in frame 2 is prepended by the REQ socket when it sends the message to the ROUTER.
- 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 3-2).

Figure 3-2. Multihop Request-reply Envelope



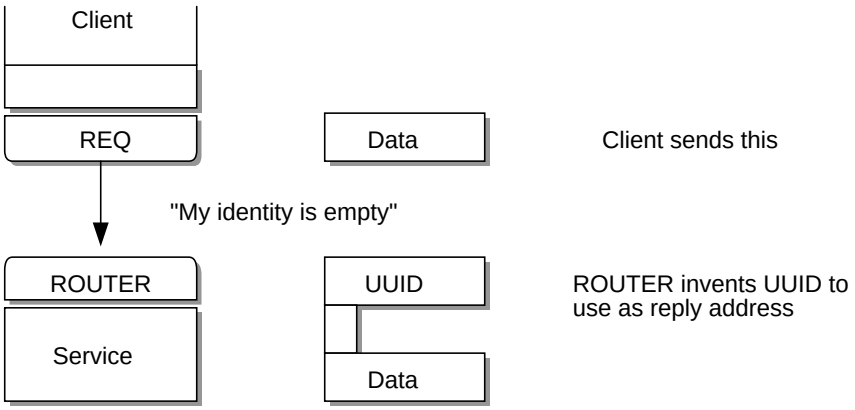
Here now is a more detailed explanation of the four socket types we use for request-reply patterns:

- DEALER just deals out the messages you send to all connected peers (aka "round-robin"), and deals in (aka "fair queuing") the messages it receives. It is exactly like a PUSH and PULL socket combined.
- REQ prepends an empty message frame to every message you send, and removes the empty message frame 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 frame) from each message it sends, and uses that reply address to decide which peer the message should go to.
- REP stores all the message frames up to the first empty message frame, 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 frame. If you’re not using REQ at the other end of the chain then you must add the empty message frame yourself.

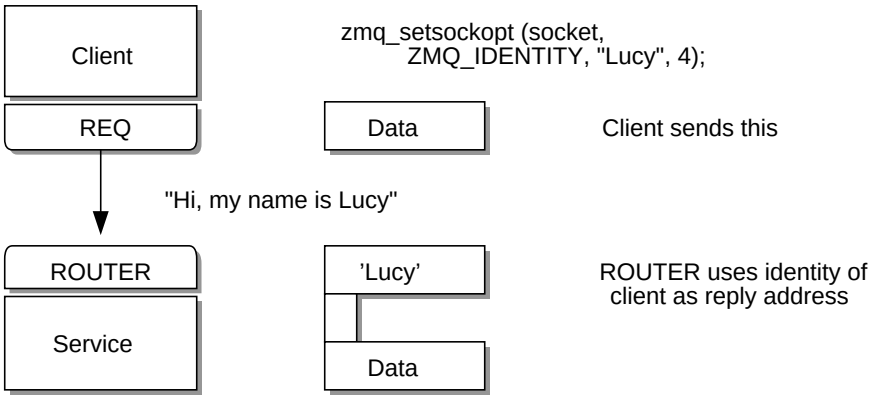
So the obvious question about ROUTER is, where does it get the reply addresses from? And the obvious answer is, it uses the socket’s identity. As we already learned, if a socket does not set an identity, the ROUTER generates an identity that it can associate with the connection to that socket(Figure 3-3).

Figure 3-3. ROUTER Invents a UUID



When we set our own identity on a socket, this gets passed to the ROUTER, which passes it to the application as part of the envelope for each message that comes in(Figure 3-4).

Figure 3-4. ROUTER uses Identity If It knows It



Let’s observe the above two cases in practice. This program dumps the contents of the message frames that a ROUTER receives from two REP sockets, one not using identities, and one using an identity ’Hello’:

Example 3-1. Identity check (identity.lua)

```
--
--  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.
--
--  Author: Robert G. Jakabosky <bobby@sharedrealm.com>
--
require"zmq"
require"zhelpers"

local context = zmq.init(1)

local sink = context:socket(zmq.ROUTER)
sink:bind("inproc://example")

--  First allow 0MQ to set the identity
local anonymous = context:socket(zmq.REQ)
anonymous:connect("inproc://example")
anonymous:send("ROUTER uses a generated UUID")
s_dump(sink)

--  Then set the identity ourself
local identified = context:socket(zmq.REQ)
identified:setopt(zmq.IDENTITY, "Hello")
identified:connect("inproc://example")
identified:send("ROUTER socket uses REQ's socket identity")
s_dump(sink)

sink:close()
anonymous:close()
identified:close()
context:term()
```

Here is what the dump function prints:

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

3.2. Custom Request-Reply Routing

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. These provide your kindergarten request-reply socket pattern. It's an easy pattern to learn but quite rapidly gets annoying as it provides, for instance, no way to resend a request if it got lost for some reason.

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 REQs and REPS 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:

- By default, a peer has a null identity and the ROUTER will generate a UUID and use that to refer to the connection when it delivers you each incoming message from that peer.
- If the peer socket set an identity, the ROUTER will give that identity when it delivers an incoming request envelope from that peer.
- 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 at least three routing patterns, one for each of the socket types we can easily connect to a ROUTER:

- ROUTER-to-DEALER.
- ROUTER-to-REQ.
- ROUTER-to-REP.

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.

3.3. ROUTER-to-DEALER Routing

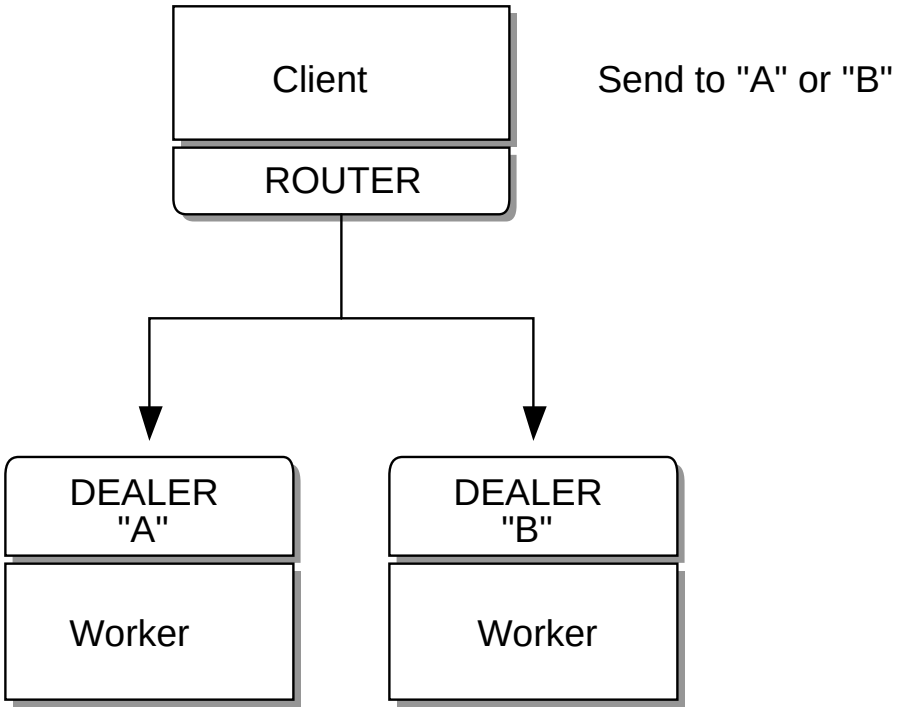
The ROUTER-to-DEALER pattern is the simplest. You connect one ROUTER to many DEALERs, and then distribute messages to the DEALERs using any algorithm you like. The DEALERs can be sinks (process the messages without any response), proxies (send the messages on to other nodes), or services (send back replies).

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 just round-robin 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 3-5).

Figure 3-5. ROUTER-to-DEALER Custom Routing



Here's code that shows how this works:

Example 3-2. ROUTER-to-DEALER (rtdealer.lua)

```
--
```

```
--  Custom routing 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.
--
--  Author: Robert G. Jakabosky <bobby@sharedrealm.com>
--
require"zmq"
require"zmq.threads"
require"zhelpers"

local pre_code = [[
    local zmq = require"zmq"
    require"zhelpers"
    --local threads = require"zmq.threads"
    --local context = threads.get_parent_ctx()
]]

--  We have two workers, here we copy the code, normally these would
--  run on different boxes...
--
local worker_task_a = pre_code .. [[
    local context = zmq.init(1)
    local worker = context:socket(zmq.DEALER)
    worker:setopt(zmq.IDENTITY, "A")
    worker:connect("ipc://routing.ipc")

    local total = 0
    while true do
        --  We receive one part, with the workload
        local request = worker:recv()
        local finished = (request == "END")

        if (finished) then
            printf ("A received: %d\n", total)
            break
        end
        total = total + 1
    end
    worker:close()
    context:term()
]]

local worker_task_b = pre_code .. [[
    local context = zmq.init(1)
    local worker = context:socket(zmq.DEALER)
    worker:setopt(zmq.IDENTITY, "B")
    worker:connect("ipc://routing.ipc")

    local total = 0
    while true do
        --  We receive one part, with the workload
```

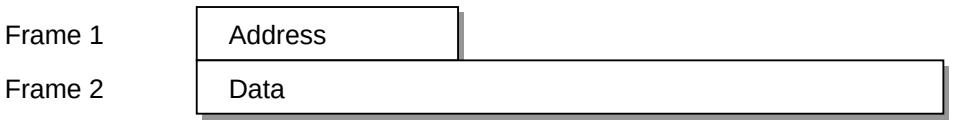
```
        local request = worker:recv()
        local finished = (request == "END")

        if (finished) then
            printf ("B received: %d\n", total)
            break
        end
        total = total + 1
    end
    worker:close()
    context:term()
]]

s_version_assert (2, 1)
local context = zmq.init(1)

local client = context:socket(zmq.ROUTER)
client:bind("ipc://routing.ipc")

local task_a = zmq.threads.runstring(context, worker_task_a)
task_a:start()

local task_b = zmq.threads.runstring(context, worker_task_b)
task_b:start()

--  Wait for threads to connect, since otherwise the messages
--  we send won't be routable.
s_sleep (1000)

--  Send 10 tasks scattered to A twice as often as B
math.randomseed(os.time())
for n=1,10 do
    --  Send two message parts, first the address...
    if (randof (3) > 0) then
        client:send("A", zmq.SNDMORE)
    else
        client:send("B", zmq.SNDMORE)
    end

    --  And then the workload
    client:send("This is the workload")
end
client:send("A", zmq.SNDMORE)
client:send("END")

client:send("B", zmq.SNDMORE)
client:send("END")

client:close()
context:term()

assert(task_a:join())
assert(task_b:join())
```

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 ROUTERs. 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 consisting of just an identity frame (we don't need a null separator)(Figure 3-6).

Figure 3-6. 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 sort from the destination node. We'll come to creating reliable patterns later on.

DEALERs in fact work exactly like PUSH and PULL combined. Do not however connect PUSH or PULL sockets to DEALERS. That would just be nasty and pointless.

3.4. Least-Recently Used Routing (LRU Pattern)

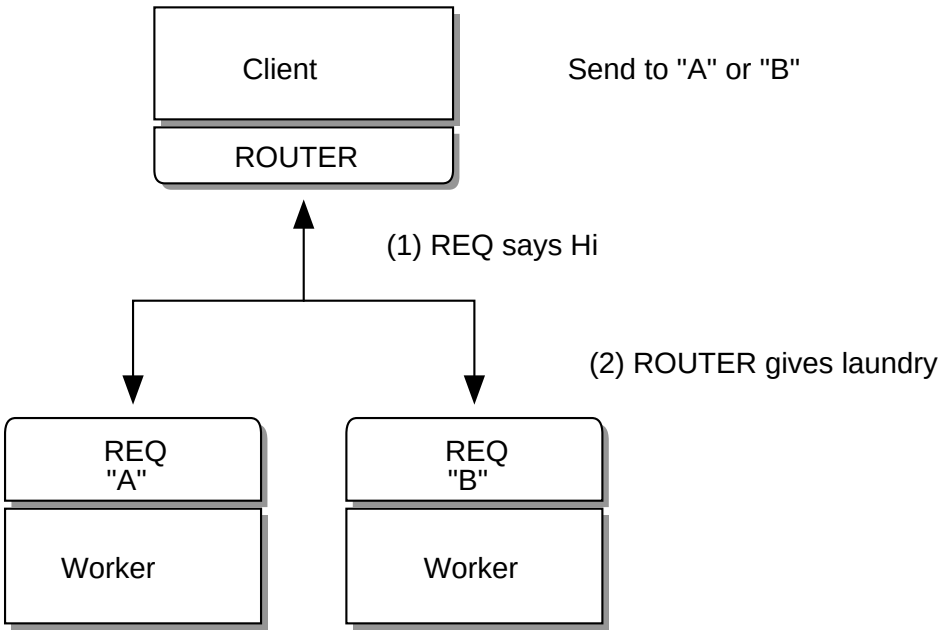
REQ sockets don't listen to you, and if you try to speak out of turn they'll ignore you. You have to wait for them to say something, and *then* you can give a sarcastic answer. This is very useful for routing because it means we can keep a bunch of REQs waiting for answers. In effect, a REQ socket will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

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

- REQ speaks to ROUTER
- ROUTER replies to REQ
- REQ speaks to ROUTER
- ROUTER replies to REQ
- etc.

Like DEALERs, REQs can only talk to one ROUTER and since REQs always start by talking to the ROUTER, you should never connect one REQ to more than one ROUTER unless you are doing sneaky stuff like multi-pathway redundant routing(Figure 3-7). 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 3-7. ROUTER to REQ Custom Routing



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

Example 3-3. ROUTER-to-REQ (rtmama.lua)

```
--
```

```
--  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.
--
--  Author: Robert G. Jakabosky <bobby@sharedrealm.com>
--
require"zmq"
require"zmq.threads"
require"zhelpers"

NBR_WORKERS  = 10

local pre_code = [[
    local identity, seed = ...
    local zmq = require"zmq"
    require"zhelpers"
    math.randomseed(seed)
]]

local worker_task = pre_code .. [[
    local context = zmq.init(1)
    local worker = context:socket(zmq.REQ)

    --  We use a string identity for ease here
    worker:setopt(zmq.IDENTITY, identity)
    worker:connect("ipc://routing.ipc")

    local total = 0
    while true do
        --  Tell the router we're ready for work
        worker:send("ready")

        --  Get workload from router, until finished
        local workload = worker:recv()
        local finished = (workload == "END")

        if (finished) then
            printf ("Processed: %d tasks\n", total)
            break
        end
        total = total + 1

        --  Do some random work
        s_sleep (randof (1000) + 1)
    end
    worker:close()
    context:term()
]]

s_version_assert (2, 1)
local context = zmq.init(1)
```

```
local client = context:socket(zmq.ROUTER)
client:bind("ipc://routing.ipc")
math.randomseed(os.time())

local workers = {}
for n=1,NBR_WORKERS do
    local identity = string.format("%04X-%04X", randof (0x10000), randof (0x10000))
    local seed = os.time() + math.random()
    workers[n] = zmq.threads.runstring(context, worker_task, identity, seed)
    workers[n]:start()
end
for n=1,(NBR_WORKERS * 10) do
    --  LRU worker is next waiting in queue
    local address = client:recv()
    local empty = client:recv()

    local ready = client:recv()

    client:send(address, zmq.SNDMORE)
    client:send("", zmq.SNDMORE)
    client:send("This is the workload")

end
--  Now ask mamas to shut down and report their results
for n=1,NBR_WORKERS do
    local address = client:recv()
    local empty = client:recv()

    local ready = client:recv()

    client:send(address, zmq.SNDMORE)
    client:send("", zmq.SNDMORE)
    client:send("END")

end

for n=1,NBR_WORKERS do
    assert(workers[n]:join())
end

client:close()
context:term()
```

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 REQs print the total tasks they each did. Since the REQs do random work, and we're not load balancing, we expect each REQ to do approximately the same amount but with random variation. And that is indeed what we see:

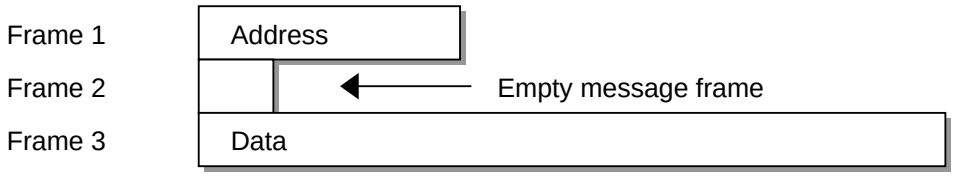
```
Processed: 8 tasks
Processed: 8 tasks
Processed: 11 tasks
Processed: 7 tasks
Processed: 9 tasks
Processed: 11 tasks
Processed: 14 tasks
Processed: 11 tasks
Processed: 11 tasks
Processed: 10 tasks
```

Some comments on this code

- We don't need any settle time, since the REQs 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 REQs would be fully anonymous and then you'd call zmq_msg_recv[3] and zmq_msg_send[3] directly instead of the zhelpers s_recv() and s_send() functions, which can only handle strings.
- 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 REQ, we must create a REQ-friendly envelope consisting of an address plus an empty message frame(Figure 3-8).

Figure 3-8. Routing Envelope for REQ



3.5. Address-based Routing

In a classic request-reply pattern a ROUTER wouldn't talk to a REP socket at all, but rather would get a DEALER to do the job for it. 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 REP and see what the heck emerges.

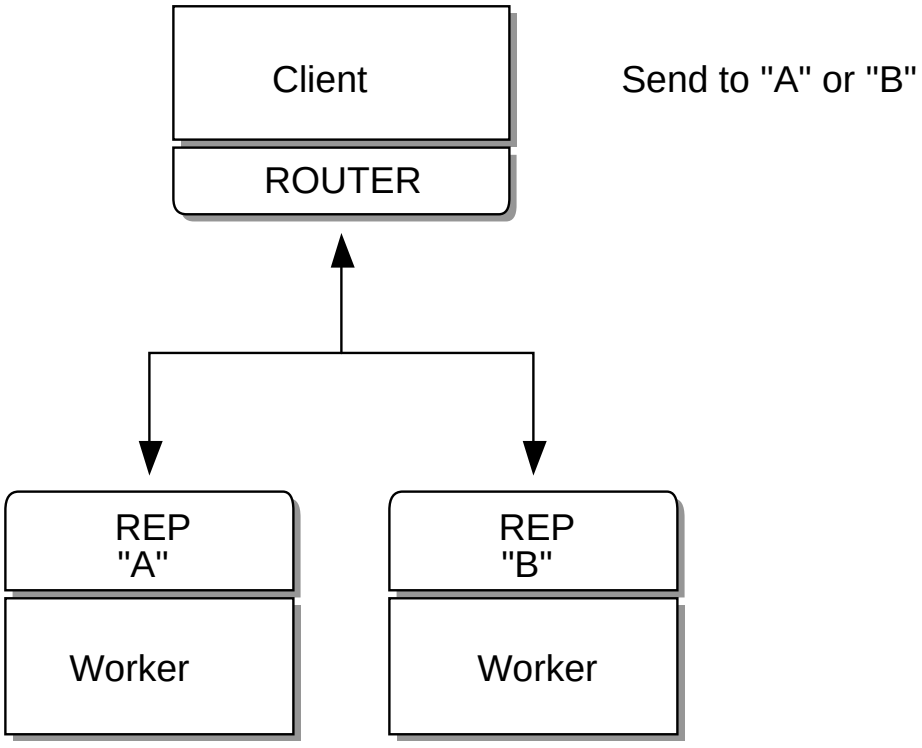
The special thing about REPs is actually two things:

- 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, REPs are anonymous and replaceable, but we're learning about custom routing. So, in our use-case we have reason to send a request to REP A rather than REP 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 REP 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 REP. 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 REP(Figure 3-9).

Figure 3-9. ROUTER-to-REP 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 REP address first.
- The ROUTER removes the REP address, uses that to decide which REP to send the message to.
- The REP 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 REP.
- The REP recreates the envelope stack and sends that back with the worker's reply to the ROUTER.
- The ROUTER prepends the REP'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 REP is garbage in, garbage out.

Example 3-4. ROUTER-to-REP (rtpapa.lua)

```
--
--  Custom routing Router to Papa (ROUTER to REP)
--
--  Author: Robert G. Jakabosky <bobby@sharedrealm.com>
--
require"zmq"
require"zhelpers"

--  We will do this all in one thread to emphasize the sequence
--  of events...

local context = zmq.init(1)

local client = context:socket(zmq.ROUTER)
client:bind("ipc://routing.ipc")

local worker = context:socket(zmq.REP)
worker:setopt(zmq.IDENTITY, "A")
worker:connect("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...
s_sleep (1000)

--  Send papa address, address stack, empty part, and request
client:send("A", zmq.SNDMORE)
client:send("address 3", zmq.SNDMORE)
client:send("address 2", zmq.SNDMORE)
client:send("address 1", zmq.SNDMORE)
client:send("", zmq.SNDMORE)
```

```
client:send("This is the workload")

--  Worker should get just the workload
s_dump (worker)

--  We don't play with envelopes in the worker
worker:send("This is the reply")

--  Now dump what we got off the ROUTER socket...
s_dump (client)

client:close()
worker:close()
context:term()
```

Run this program and it should show you this:

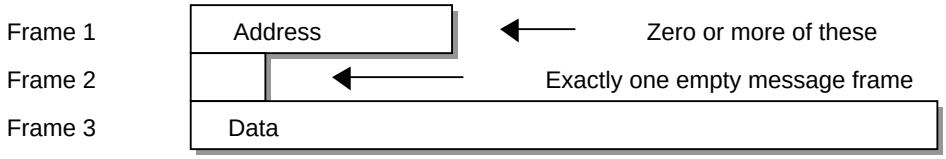
```
----------------------------------------
[020] This is the workload
----------------------------------------
[001] A
[009] address 3
[009] address 2
[009] address 1
[000]
[017] This is the reply
```

Some comments on this code:

- In reality we'd have the REP 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 REP 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 REP 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 REP socket won't get the message. (Try it.)
- We're routing using the REP's identity. Just to convince yourself this really is happening, try sending to a wrong address, like "B". The REP 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 REP, we must create a REP-friendly envelope(Figure 3-10).

Figure 3-10. Routing Envelope for REP

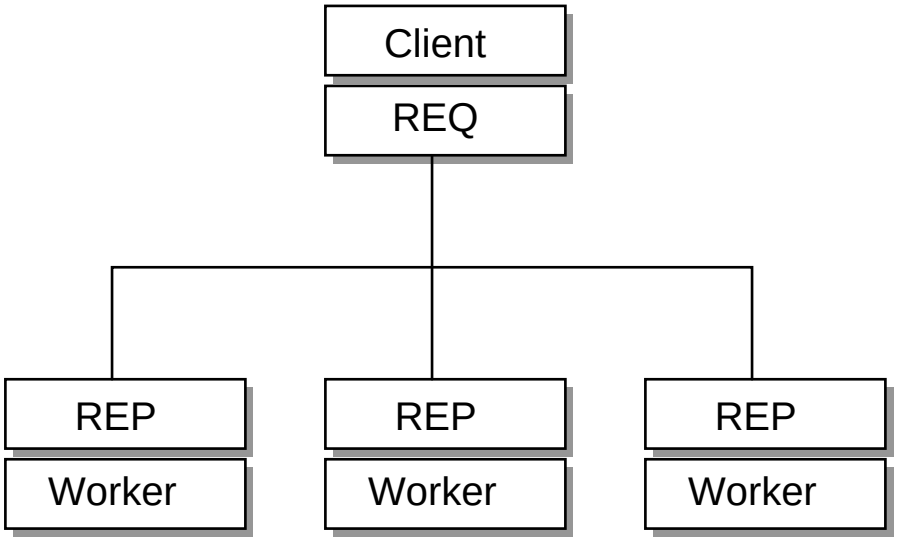


3.6. A Request-Reply Message Broker

I'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. The algorith we'll implement is *least-recently used*, since it's the most obvious use-case after simple round-robin distribution.

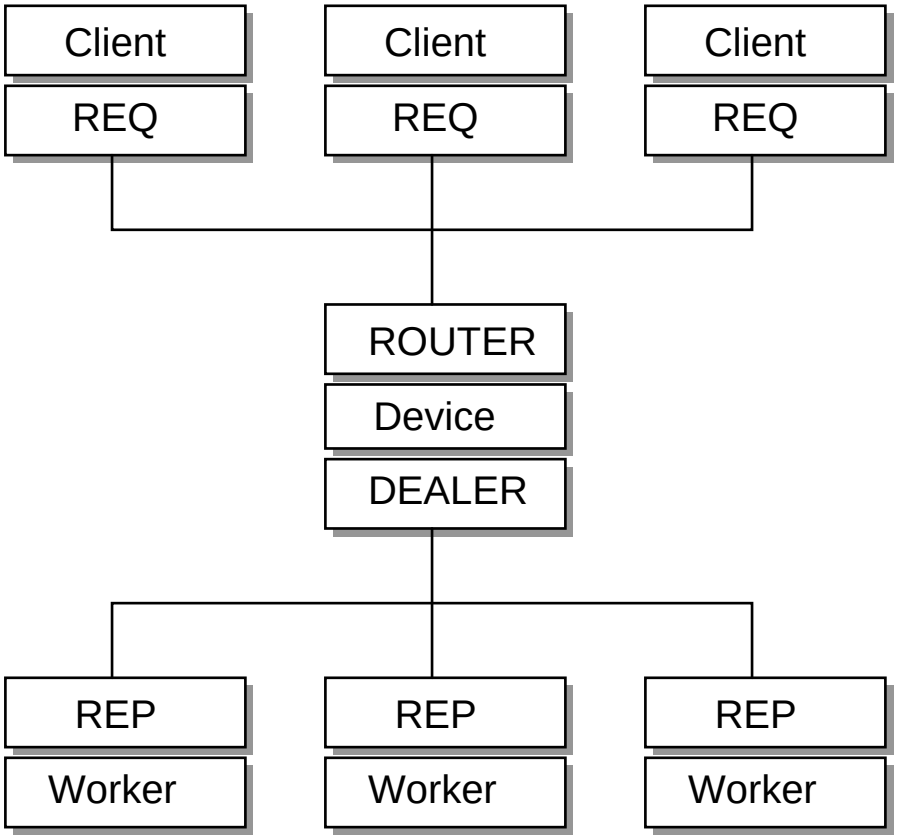
To start with, let's look back at the classic request-reply pattern and then see how it extends over a larger and larger service-oriented network. The basic pattern just has one client talking to a few workers(Figure 3-11).

Figure 3-11. Basic Request-reply



This extends to multiple workers, but if we want to handle multiple clients as well, we need a device in the middle. We'd use a simple ZMQ_QUEUE device connecting a ROUTER and a DEALER back to back. This device just switches message frames between the two sockets as fast as it can(Figure 3-12).

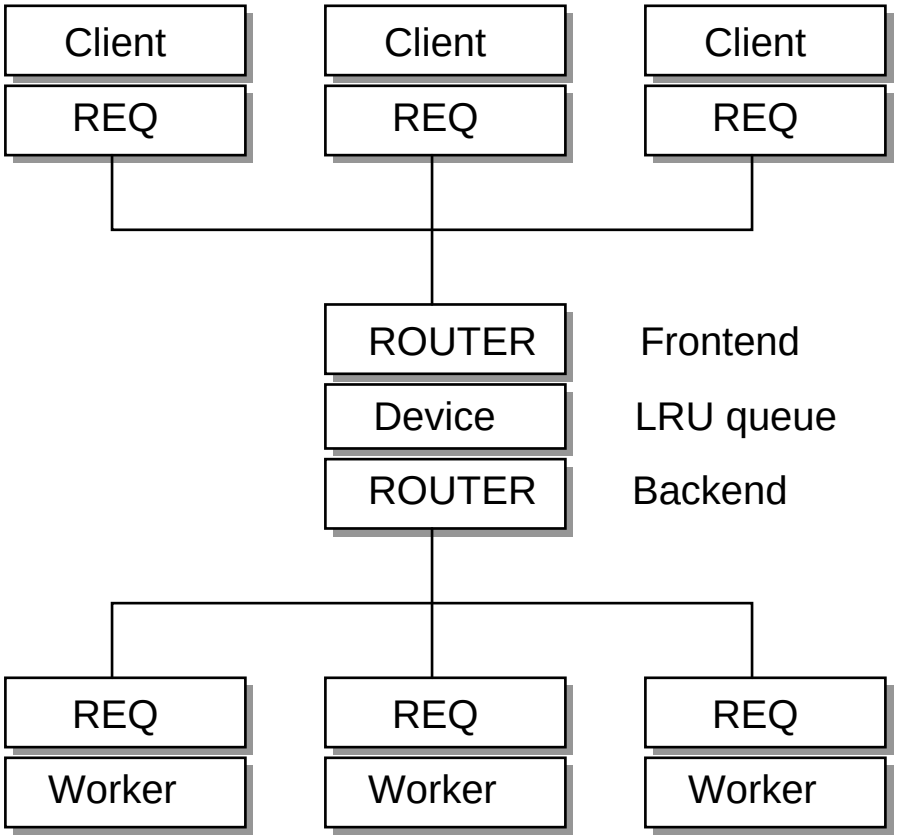
Figure 3-12. Stretched Request-reply



The key here is that the ROUTER stores the originating client address in the request envelope, the DEALER and workers don't touch that, and so the ROUTER knows which client to send the reply back to. This pattern assumes all workers provide the exact same service.

In the above design, we're using the built-in round-robin routing that DEALER provides. However this means some workers may be idle while others have multiple requests waiting. For better efficiency and proper load-balancing we want to use a least-recently used algorithm, so we take the ROUTER-REQ pattern we learned, and apply that(Figure 3-13).

Figure 3-13. Stretched Request-reply with LRU



Our broker - a ROUTER-to-ROUTER LRU queue - can't simply copy message frames blindly. Here is the code, it's a fair chunk of code, but we can reuse the core logic any time we want to do load-balancing:

Example 3-5. LRU queue broker (lruqueue.lua)

```
--
--  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.
--
--  Author: Robert G. Jakabosky <bobby@sharedrealm.com>
--
```

```
require"zmq"
require"zmq.threads"
require"zmq.poller"
require"zhelpers"

local tremove = table.remove

local NBR_CLIENTS  = 10
local NBR_WORKERS  = 3

local pre_code = [[
    local identity, seed = ...
    local zmq = require"zmq"
    require"zhelpers"
    math.randomseed(seed)
]]

--  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.
--
local client_task = pre_code .. [[
    local context = zmq.init(1)
    local client = context:socket(zmq.REQ)
    client:setopt(zmq.IDENTITY, identity)  --  Set a printable identity
    client:connect("ipc://frontend.ipc")

    --  Send request, get reply
    client:send("HELLO")
    local reply = client:recv()
    printf ("Client: %s\n", reply)

    client:close()
    context:term()
]]

--  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.
--
local worker_task = pre_code .. [[
    local context = zmq.init(1)
    local worker = context:socket(zmq.REQ)
    worker:setopt(zmq.IDENTITY, identity)  --  Set a printable identity
    worker:connect("ipc://backend.ipc")

    --  Tell broker we're ready for work
    worker:send("READY")

    while true do
        --  Read and save all frames until we get an empty frame
        --  In this example there is only 1 but it could be more
        local address = worker:recv()
```

```
        local empty = worker:recv()
        assert (#empty == 0)

        --  Get request, send reply
        local request = worker:recv()
        printf ("Worker: %s\n", request)

        worker:send(address, zmq.SNDMORE)
        worker:send("", zmq.SNDMORE)
        worker:send("OK")

    end
    worker:close()
    context:term()
]]

s_version_assert (2, 1)

--  Prepare our context and sockets
local context = zmq.init(1)
local frontend = context:socket(zmq.ROUTER)
local backend  = context:socket(zmq.ROUTER)
frontend:bind("ipc://frontend.ipc")
backend:bind("ipc://backend.ipc")

local clients = {}
for n=1,NBR_CLIENTS do
    local identity = string.format("%04X-%04X", randof (0x10000), randof (0x10000))
    local seed = os.time() + math.random()
    clients[n] = zmq.threads.runstring(context, client_task, identity, seed)
    clients[n]:start()
end
local workers = {}
for n=1,NBR_WORKERS do
    local identity = string.format("%04X-%04X", randof (0x10000), randof (0x10000))
    local seed = os.time() + math.random()
    workers[n] = zmq.threads.runstring(context, worker_task, identity, seed)
    workers[n]:start(true)
end

--  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
local worker_queue = {}

local is_accepting = false
local max_requests = #clients

local poller = zmq.poller(2)
```

```
local function frontend_cb()
    --  Now get next client request, route to LRU worker
    --  Client request is [address][empty][request]
    local client_addr = frontend:recv()
    local empty = frontend:recv()
    assert (#empty == 0)

    local request = frontend:recv()

    -- Dequeue a worker from the queue.
    local worker = tremove(worker_queue, 1)

    backend:send(worker, zmq.SNDMORE)
    backend:send("", zmq.SNDMORE)
    backend:send(client_addr, zmq.SNDMORE)
    backend:send("", zmq.SNDMORE)
    backend:send(request)

    if (#worker_queue == 0) then
        -- stop accepting work from clients, when no workers are available.
        poller:remove(frontend)
        is_accepting = false
    end
end

poller:add(backend, zmq.POLLIN, function()
    --  Queue worker address for LRU routing
    local worker_addr = backend:recv()
    worker_queue[#worker_queue + 1] = worker_addr

    -- start accepting client requests, if we are not already doing so.
    if not is_accepting then
        is_accepting = true
        poller:add(frontend, zmq.POLLIN, frontend_cb)
    end

    --  Second frame is empty
    local empty = backend:recv()
    assert (#empty == 0)

    --  Third frame is READY or else a client reply address
    local client_addr = backend:recv()

    --  If client reply, send rest back to frontend
    if (client_addr ~= "READY") then
        empty = backend:recv()
        assert (#empty == 0)

        local reply = backend:recv()
        frontend:send(client_addr, zmq.SNDMORE)
        frontend:send("", zmq.SNDMORE)
        frontend:send(reply)
```

```
        max_requests = max_requests - 1
        if (max_requests == 0) then
            poller:stop()      --  Exit after N messages
        end
    end
end)

-- start poller's event loop
poller:start()

frontend:close()
backend:close()
context:term()

for n=1,NBR_CLIENTS do
    assert(clients[n]:join())
end
-- workers are detached, we don't need to join with them.
```

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 REQ 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 REQ 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 this code we set the identity of client and worker sockets to make it easier to trace the message frames. Most normal applications do not use identities. Let's assume the client's identity is "CLIENT" and the worker's identity is "WORKER". The client sends a single frame with the message(Figure 3-14).

Figure 3-14. Message that Client Sends

What the queue gets, when reading off the ROUTER frontend socket, are three frames consisting of the sender address, empty frame delimiter, and the data part(Figure 3-15).

Figure 3-15. 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 REQ at the other end happy(Figure 3-16).

Figure 3-16. Message Sent to Backend

This complex envelope stack gets chewed up first by the backend ROUTER socket, which removes the first frame. Then the REQ socket in the worker removes the empty part, and provides the rest to the worker application(Figure 3-17).

Figure 3-17. 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 frame) 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 REQ 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.

3.7. A High-Level API for ØMQ

Reading and writing multi-part messages using the native ØMQ API is, to be polite, a lot of work. 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_msg_recv (worker, &address, 0);

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

    //  Get request, send reply
    zmq_msg_t payload;
    zmq_msg_init (&payload);
    zmq_msg_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_msg_send (worker, &address, ZMQ_SNDMORE);
    zmq_close (&address);
    zmq_msg_send (worker, &empty, ZMQ_SNDMORE);
    zmq_close (&empty);
    zmq_msg_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 multi-part 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 frame), 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 (http://zero.mq/c), 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. CZMQ also uses an elegant object model that leads to frankly lovely code.

Here is the LRU queue broker rewritten to use CZMQ:

Example 3-6. LRU queue broker using CZMQ (lruqueue2.lua)

```
--
--  Least-recently used (LRU) queue device
--  Demonstrates use of the msg class
--
--  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.
--
--  Author: Robert G. Jakabosky <bobby@sharedrealm.com>
```

```
--

require"zmq"
require"zmq.threads"
require"zmq.poller"
require"zmsg"

local tremove = table.remove

local NBR_CLIENTS  = 10
local NBR_WORKERS  = 3

local pre_code = [[
    local identity, seed = ...
    local zmq = require"zmq"
    local zmsg = require"zmsg"
    require"zhelpers"
    math.randomseed(seed)
]]

--  Basic request-reply client using REQ socket
--
local client_task = pre_code .. [[
    local context = zmq.init(1)
    local client = context:socket(zmq.REQ)
    client:setopt(zmq.IDENTITY, identity)  --  Set a printable identity
    client:connect("ipc://frontend.ipc")

    --  Send request, get reply
    client:send("HELLO")
    local reply = client:recv()
    printf ("Client: %s\n", reply)

    client:close()
    context:term()
]]

--  Worker using REQ socket to do LRU routing
--
local worker_task = pre_code .. [[
    local context = zmq.init(1)
    local worker = context:socket(zmq.REQ)
    worker:setopt(zmq.IDENTITY, identity)  --  Set a printable identity
    worker:connect("ipc://backend.ipc")

    --  Tell broker we're ready for work
    worker:send("READY")

    while true do
        local msg = zmsg.recv (worker)
        printf ("Worker: %s\n", msg:body())
        msg:body_set("OK")
        msg:send(worker)
```

```
    end
    worker:close()
    context:term()
]]

s_version_assert (2, 1)

--  Prepare our context and sockets
local context = zmq.init(1)
local frontend = context:socket(zmq.ROUTER)
local backend  = context:socket(zmq.ROUTER)
frontend:bind("ipc://frontend.ipc")
backend:bind("ipc://backend.ipc")

local clients = {}
for n=1,NBR_CLIENTS do
    local identity = string.format("%04X-%04X", randof (0x10000), randof (0x10000))
    local seed = os.time() + math.random()
    clients[n] = zmq.threads.runstring(context, client_task, identity, seed)
    clients[n]:start()
end
local workers = {}
for n=1,NBR_WORKERS do
    local identity = string.format("%04X-%04X", randof (0x10000), randof (0x10000))
    local seed = os.time() + math.random()
    workers[n] = zmq.threads.runstring(context, worker_task, identity, seed)
    workers[n]:start(true)
end

--  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
local worker_queue = {}

local is_accepting = false
local max_requests = #clients

local poller = zmq.poller(2)

local function frontend_cb()
    --  Now get next client request, route to next worker
    local msg = zmsg.recv (frontend)

    -- Dequeue a worker from the queue.
    local worker = tremove(worker_queue, 1)

    msg:wrap(worker, "")
    msg:send(backend)
```

```
    if (#worker_queue == 0) then
        -- stop accepting work from clients, when no workers are available.
        poller:remove(frontend)
        is_accepting = false
    end
end

poller:add(backend, zmq.POLLIN, function()
    local msg = zmsg.recv(backend)
    --  Use worker address for LRU routing
    worker_queue[#worker_queue + 1] = msg:unwrap()

    -- start accepting client requests, if we are not already doing so.
    if not is_accepting then
        is_accepting = true
        poller:add(frontend, zmq.POLLIN, frontend_cb)
    end

    --  Forward message to client if it's not a READY
    if (msg:address() ~= "READY") then
        msg:send(frontend)

        max_requests = max_requests - 1
        if (max_requests == 0) then
            poller:stop()      --  Exit after N messages
        end
    end
end)

-- start poller's event loop
poller:start()

frontend:close()
backend:close()
context:term()

for n=1,NBR_CLIENTS do
    assert(clients[n]:join())
end
-- workers are detached, we don't need to join with them.
```

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 will return NULL in such cases. 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.
- Cancel a timer.

`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`:

Example 3-7. LRU queue broker using zloop (lruqueue3.lua)

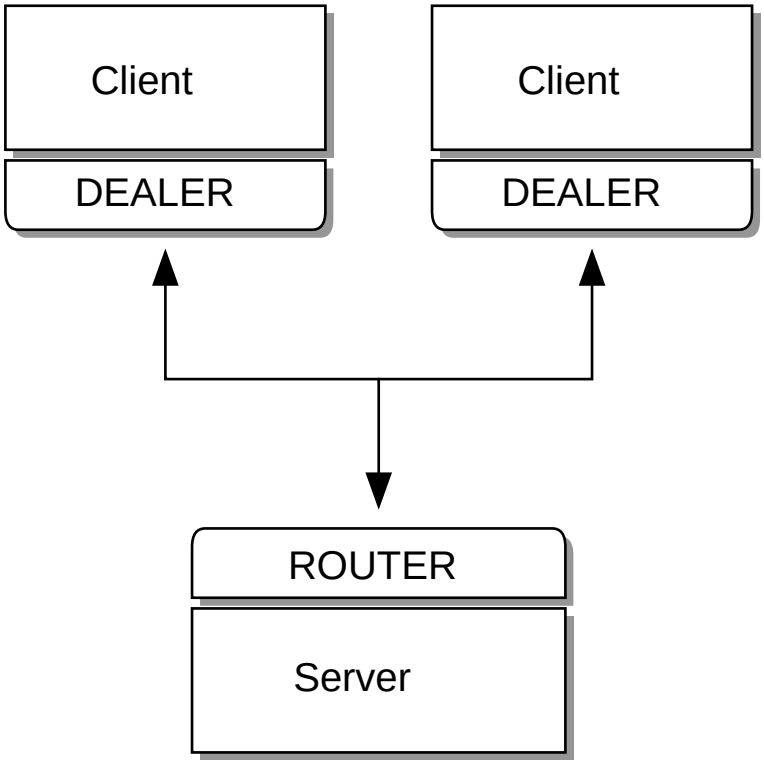
```
(This example still needs translation into Lua)
```

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`.

3.8. Asynchronous Client-Server

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 3-18).

Figure 3-18. 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:

Example 3-8. Asynchronous client-server (asyncsrv.lua)

```
--
--  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.
--
--  Author: Robert G. Jakabosky <bobby@sharedrealm.com>
--

require"zmq"
require"zmq.threads"
require"zmsg"
require"zhelpers"

local NBR_CLIENTS  = 3

--  ---------------------------------------------------------------------
--  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.

local client_task = [[
    local identity, seed = ...
    local zmq = require"zmq"
    require"zmq.poller"
    require"zmq.threads"
    local zmsg = require"zmsg"
    require"zhelpers"
    math.randomseed(seed)

    local context = zmq.init(1)
    local client = context:socket(zmq.DEALER)

    --  Generate printable identity for the client
    client:setopt(zmq.IDENTITY, identity)
    client:connect("tcp://localhost:5570")

    local poller = zmq.poller(2)

    poller:add(client, zmq.POLLIN, function()
        local msg = zmsg.recv (client)
        printf ("%s: %s\n", identity, msg:body())
    end)
    local request_nbr = 0
    while true do
        --  Tick once per second, pulling in arriving messages
        local centitick
        for centitick=1,100 do
            poller:poll(10000)
```

```
        end
        local msg = zmsg.new()
        request_nbr = request_nbr + 1
        msg:body_fmt("request #%d", request_nbr)
        msg:send(client)
    end
    --  Clean up and end task properly
    client:close()
    context:term()
]]

--  ---------------------------------------------------------------------
--  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.

local server_task = [[
    local server_worker = ...
    local zmq = require"zmq"
    require"zmq.poller"
    require"zmq.threads"
    local zmsg = require"zmsg"
    require"zhelpers"
    math.randomseed(os.time())

    local context = zmq.init(1)

    --  Frontend socket talks to clients over TCP
    local frontend = context:socket(zmq.ROUTER)
    frontend:bind("tcp://*:5570")

    --  Backend socket talks to workers over inproc
    local backend = context:socket(zmq.DEALER)
    backend:bind("inproc://backend")

    --  Launch pool of worker threads, precise number is not critical
    local workers = {}
    for n=1,5 do
        local seed = os.time() + math.random()
        workers[n] = zmq.threads.runstring(context, server_worker, seed)
        workers[n]:start()
    end
    --  Connect backend to frontend via a queue device
    --  We could do this:
    --      zmq:device(.QUEUE, frontend, backend)
    --  But doing it ourselves means we can debug this more easily

    local poller = zmq.poller(2)

    poller:add(frontend, zmq.POLLIN, function()
        local msg = zmsg.recv (frontend)
```

```
        --print ("Request from client:")
        --msg:dump()
        msg:send(backend)
    end)
    poller:add(backend, zmq.POLLIN, function()
        local msg = zmsg.recv (backend)
        --print ("Reply from worker:")
        --msg:dump()
        msg:send(frontend)
    end)
    --  Switch messages between frontend and backend
    poller:start()

    for n=1,5 do
        assert(workers[n]:join())
    end
    frontend:close()
    backend:close()
    context:term()
]]

--  Accept a request and reply with the same text a random number of
--  times, with random delays between replies.
--
local server_worker = [[
    local seed = ...
    local zmq = require"zmq"
    require"zmq.threads"
    local zmsg = require"zmsg"
    require"zhelpers"
    math.randomseed(seed)
    local threads = require"zmq.threads"
    local context = threads.get_parent_ctx()

    local worker = context:socket(zmq.DEALER)
    worker:connect("inproc://backend")

    while true do
        --  The DEALER socket gives us the address envelope and message
        local msg = zmsg.recv (worker)
        assert (msg:parts() == 2)

        --  Send 0..4 replies back
        local reply
        local replies = randof (5)
        for reply=1,replies do
            --  Sleep for some fraction of a second
            s_sleep (randof (1000) + 1)
            local dup = msg:dup()
            dup:send(worker)
        end
    end
    worker:close()
```

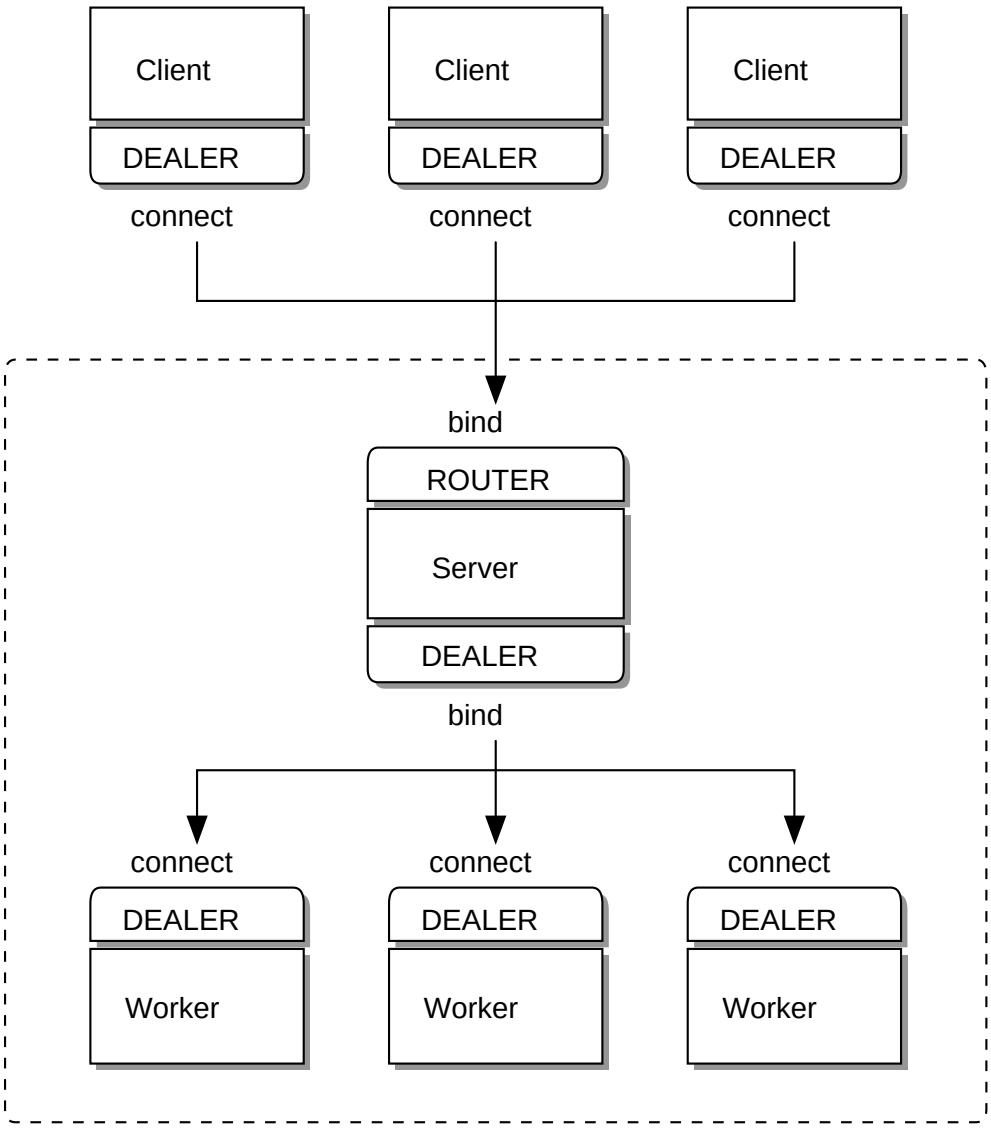
```
]]

--  This main thread simply starts several clients, and a server, and then
--  waits for the server to finish.
--

s_version_assert (2, 1)

local clients = {}
for n=1,NBR_CLIENTS do
    local identity = string.format("%04X", randof (0x10000))
    local seed = os.time() + math.random()
    clients[n] = zmq.threads.runstring(nil, client_task, identity, seed)
    clients[n]:start()
end

local server = zmq.threads.runstring(nil, server_task, server_worker)
assert(server:start())
assert(server:join())
```

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(Figure 3-19). 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.

Figure 3-19. 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 out of resources. Even if the same clients keep connecting, if you're using default identities, 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 have to:

- 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 (whether generated or explicit) as key.
- 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.

3.9. Worked Example: Inter-Broker Routing

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.

3.9.1. Establishing the Details

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 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?", we ask. 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.

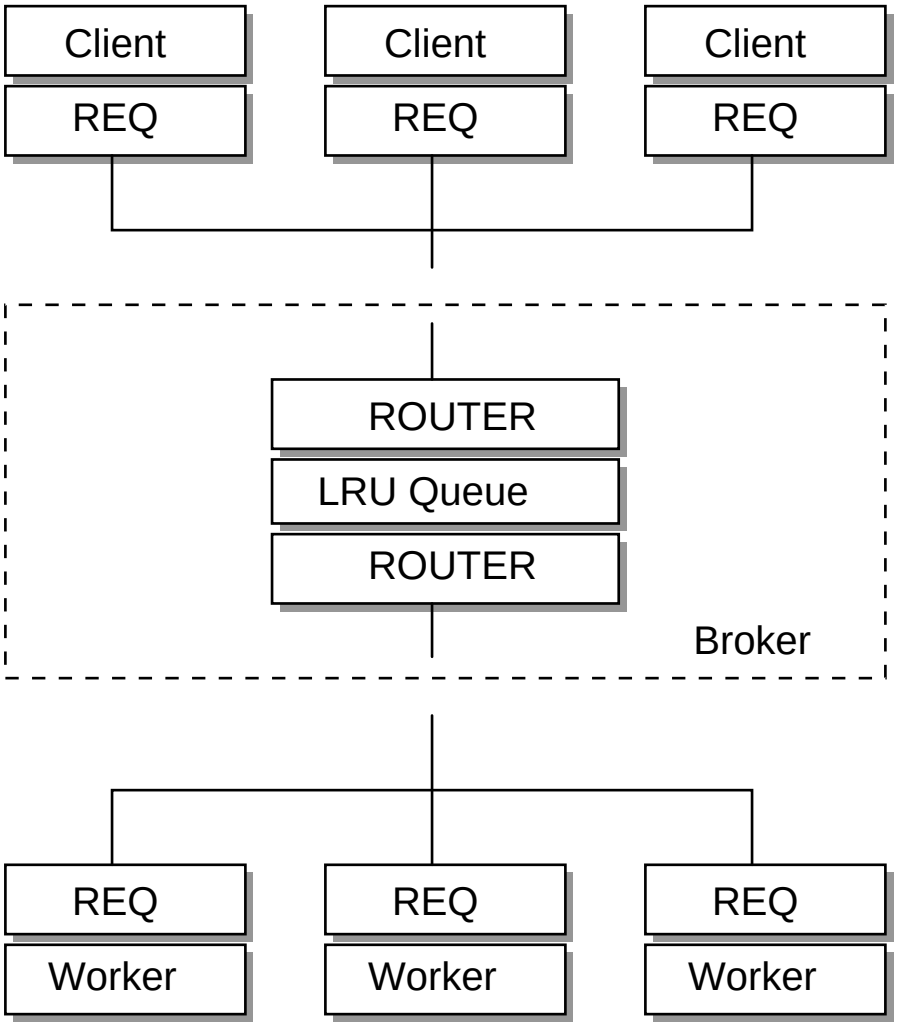
It's a straight-forward problem that requires no exotic hardware or protocols, just some clever routing algorithms and careful design. We start by designing one cluster (one data center) and then we figure out how to connect clusters together.

3.9.2. Architecture of a Single Cluster

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 3-20).

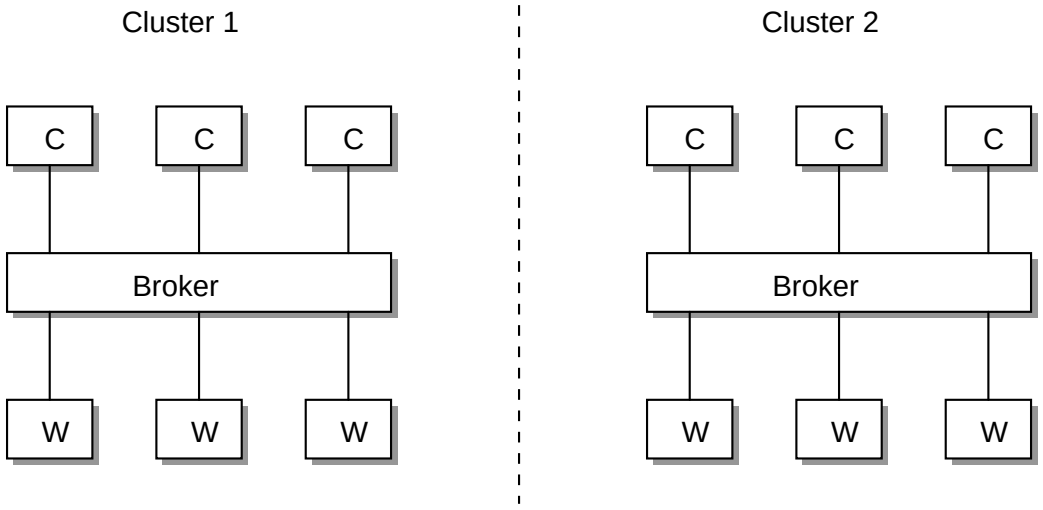
Figure 3-20. Cluster Architecture



3.9.3. Scaling to Multiple Clusters

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 3-21. Multiple Clusters

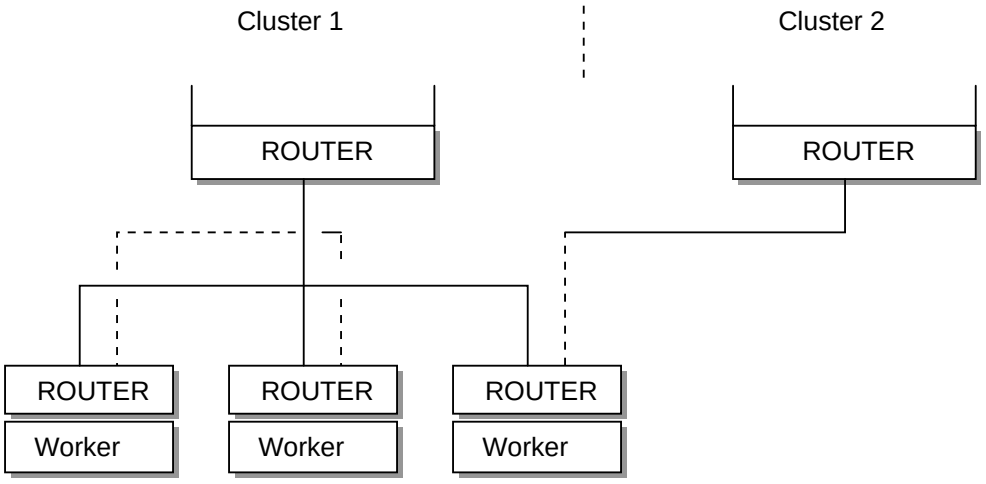


The question is: how do we get the clients of each cluster talking to the workers of the other cluster? There are a few possibilities, each with pros and cons:

- 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 fail-over logic into the clients and that's not nice.
- Workers might connect directly to both brokers. But REQ workers can't do that, they can only reply to one broker. We might use REPs but REPs 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. In this model we have workers connecting to both brokers and accepting jobs from either(Figure 3-22).

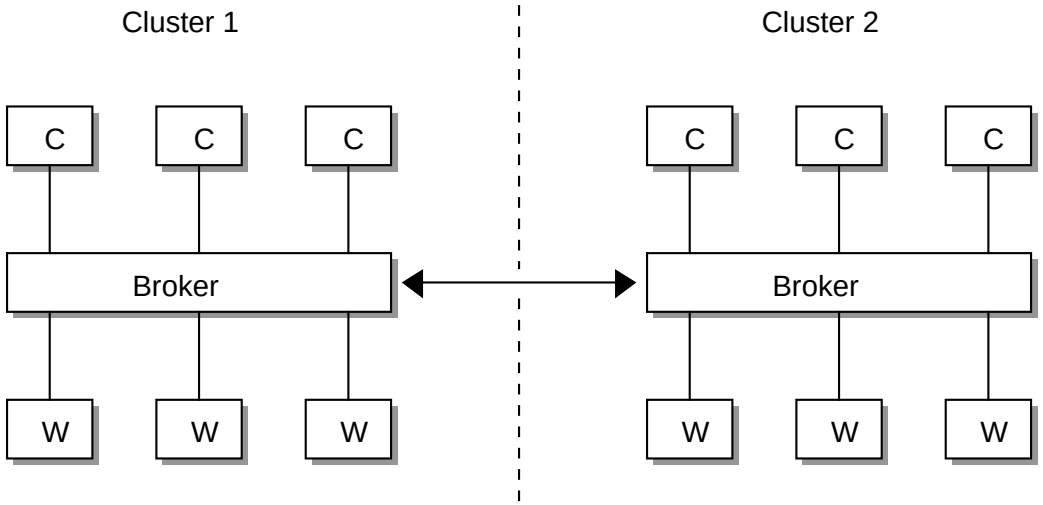
Figure 3-22. Idea 1 - Cross-connected Workers



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 REQs like we're used to(Figure 3-23).

Figure 3-23. Idea 2 - Brokers Talking to Each Other



This design is appealing because the problem is solved in one place, invisible 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. Threads 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 REQ client (REQ) threads create workloads and pass them to the broker (ROUTER).
- The REQ worker (REQ) threads process workloads and return the results to the broker (ROUTER).

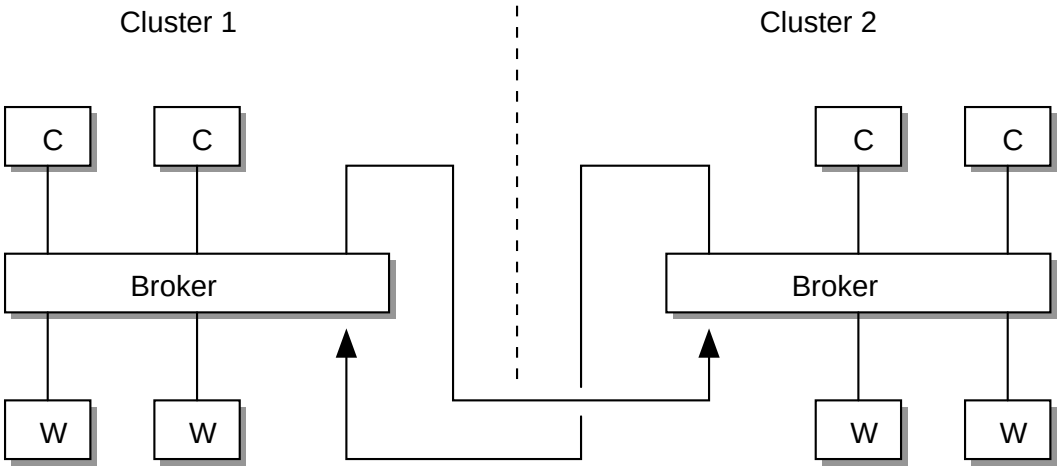
- The broker queues and distributes workloads using the LRU routing model.

3.9.4. Federation vs. Peering

There are several possible ways to interconnect 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(Figure 3-24). Note that it is legal to both bind a socket to an endpoint and connect it to other endpoints.

Figure 3-24. 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 round-robin 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.

3.9.5. The Naming Ceremony

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 nicely. It's not a big thing, but attention to details helps. 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).

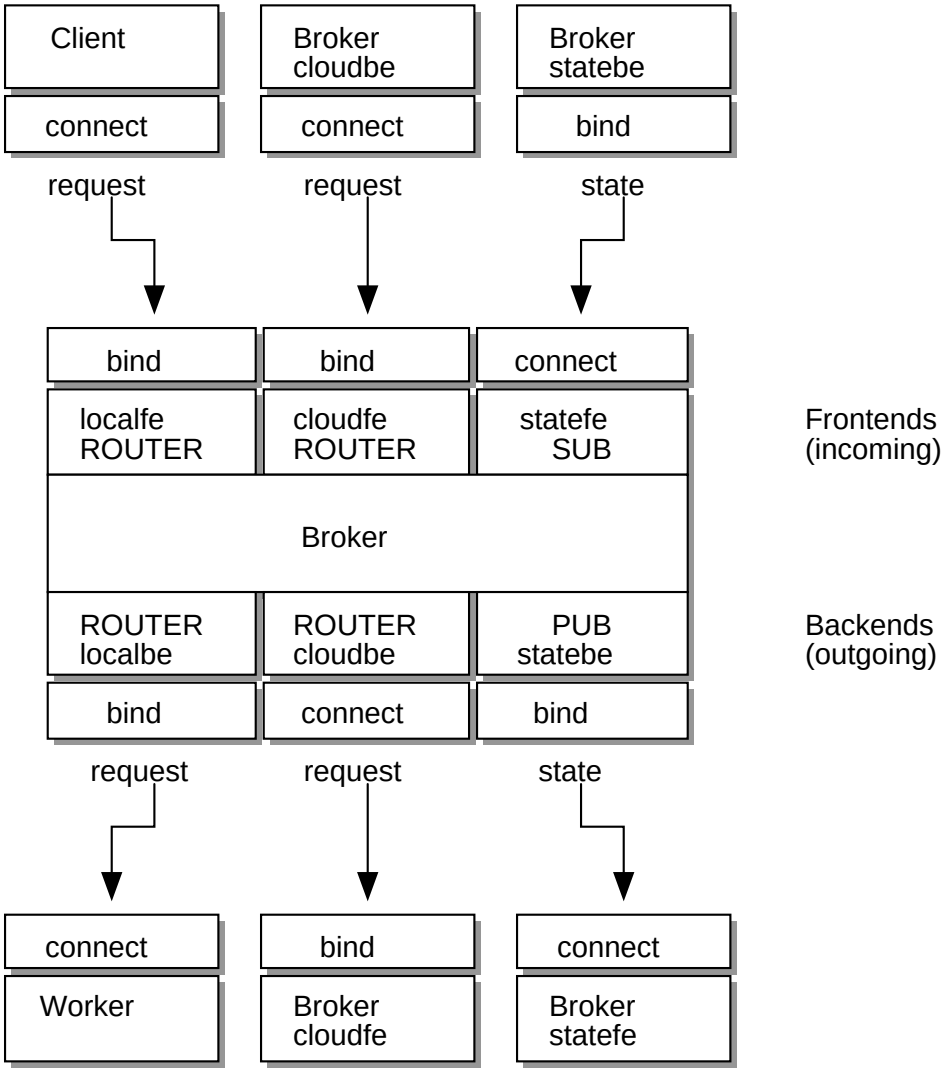
So in all the code we write for this tutorial will use these socket names:

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

For our transport and because we're simulating the whole thing on one box, 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's easier to remember "three flows, two directions" than "six different sockets"(Figure 3-25).

Figure 3-25. Broker Socket Arrangement



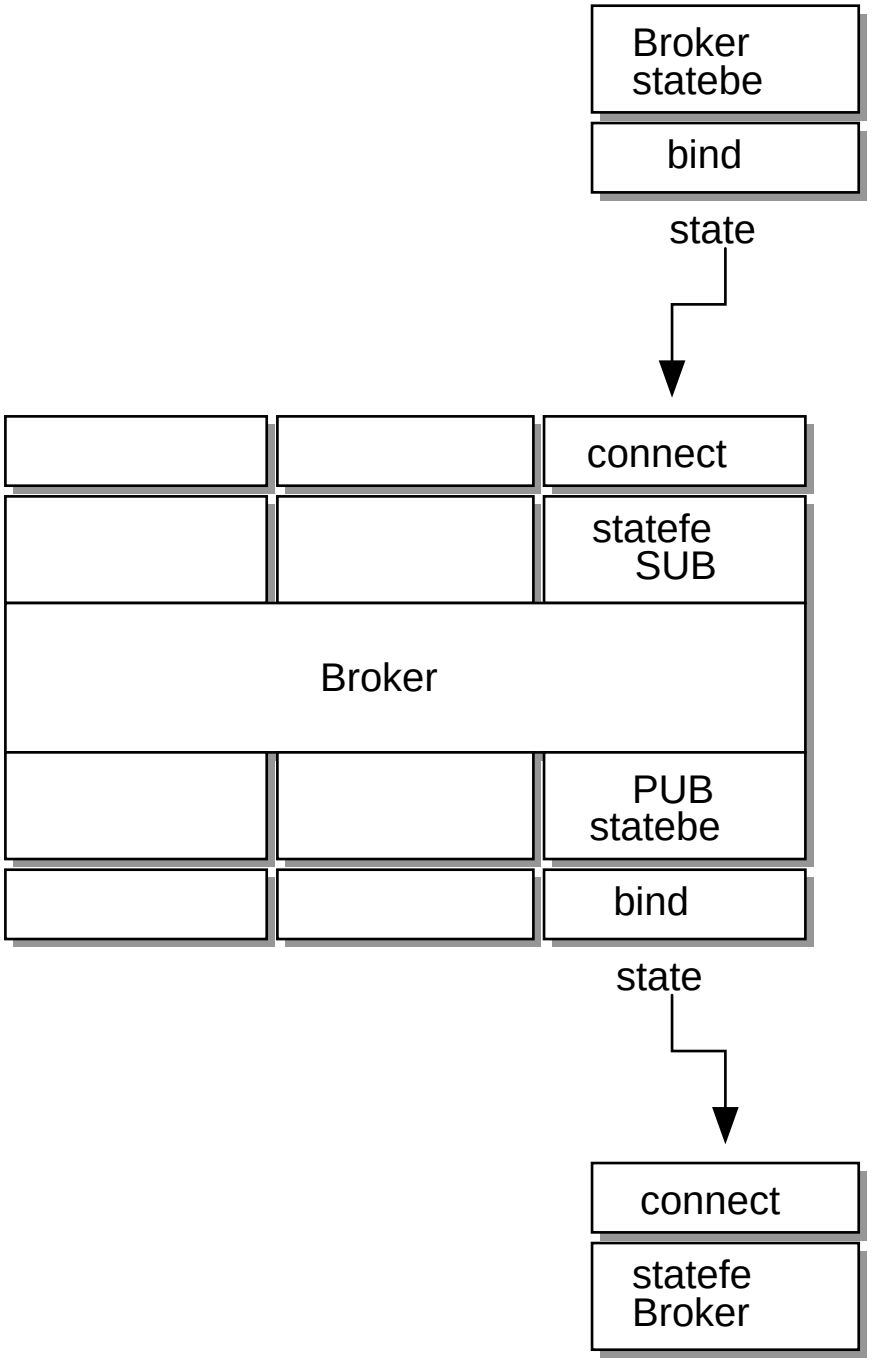
Note that we connect the cloudbe in each broker to the cloudfe in every other broker, and likewise we connect the statebe in each broker to the statefe in every other broker.

3.9.6. Prototyping the State Flow

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 3-26).

Figure 3-26. The State Flow



Here is how this works in code:

Example 3-9. Prototype state flow (peering1.lua)

```
--
--  Broker peering simulation (part 1)
--  Prototypes the state flow
--
--  Author: Robert G. Jakabosky <bobby@sharedrealm.com>
--
require"zmq"
require"zmq.poller"
require"zmsg"

--  First argument is this broker's name
--  Other arguments are our peers' names
--
if (#arg < 1) then
    printf ("syntax: peering1 me doyouend...\n")
    os.exit(-1)
end
local self = arg[1]
printf ("I: preparing broker at %s...\n", self)
math.randomseed(os.time())

--  Prepare our context and sockets
local context = zmq.init(1)

--  Bind statebe to endpoint
local statebe = context:socket(zmq.PUB)
local endpoint = string.format("ipc://%s-state.ipc", self)
assert(statebe:bind(endpoint))

--  Connect statefe to all peers
local statefe = context:socket(zmq.SUB)
statefe:setopt(zmq.SUBSCRIBE, "", 0)

for n=2,#arg do
    local peer = arg[n]
    printf ("I: connecting to state backend at '%s'\n", peer)
    local endpoint = string.format("ipc://%s-state.ipc", peer)
    assert(statefe:connect(endpoint))
end

local poller = zmq.poller(1)
--  Send out status messages to peers, and collect from peers
--  The zmq_poll timeout defines our own heartbeating
--
poller:add(statefe, zmq.POLLIN, function()
    local msg = zmsg.recv (statefe)
    printf ("%s - %s workers free\n",
        msg:address(), msg:body())
end)
```

```
while true do
    --  Poll for activity, or 1 second timeout
    local count = assert(poller:poll(1000000))

    -- if no other activity.
    if count == 0 then
        --  Send random value for worker availability
        local msg = zmsg.new()
        msg:body_fmt("%d", randof (10))
        --  We stick our own address onto the envelope
        msg:wrap(self, nil)
        msg:send(statebe)
    end
end
--  We never get here but clean up anyhow
statebe:close()
statefe:close()
context:term()
```

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, but if we were using ØMQ/2.x that would be a wise idea.

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.

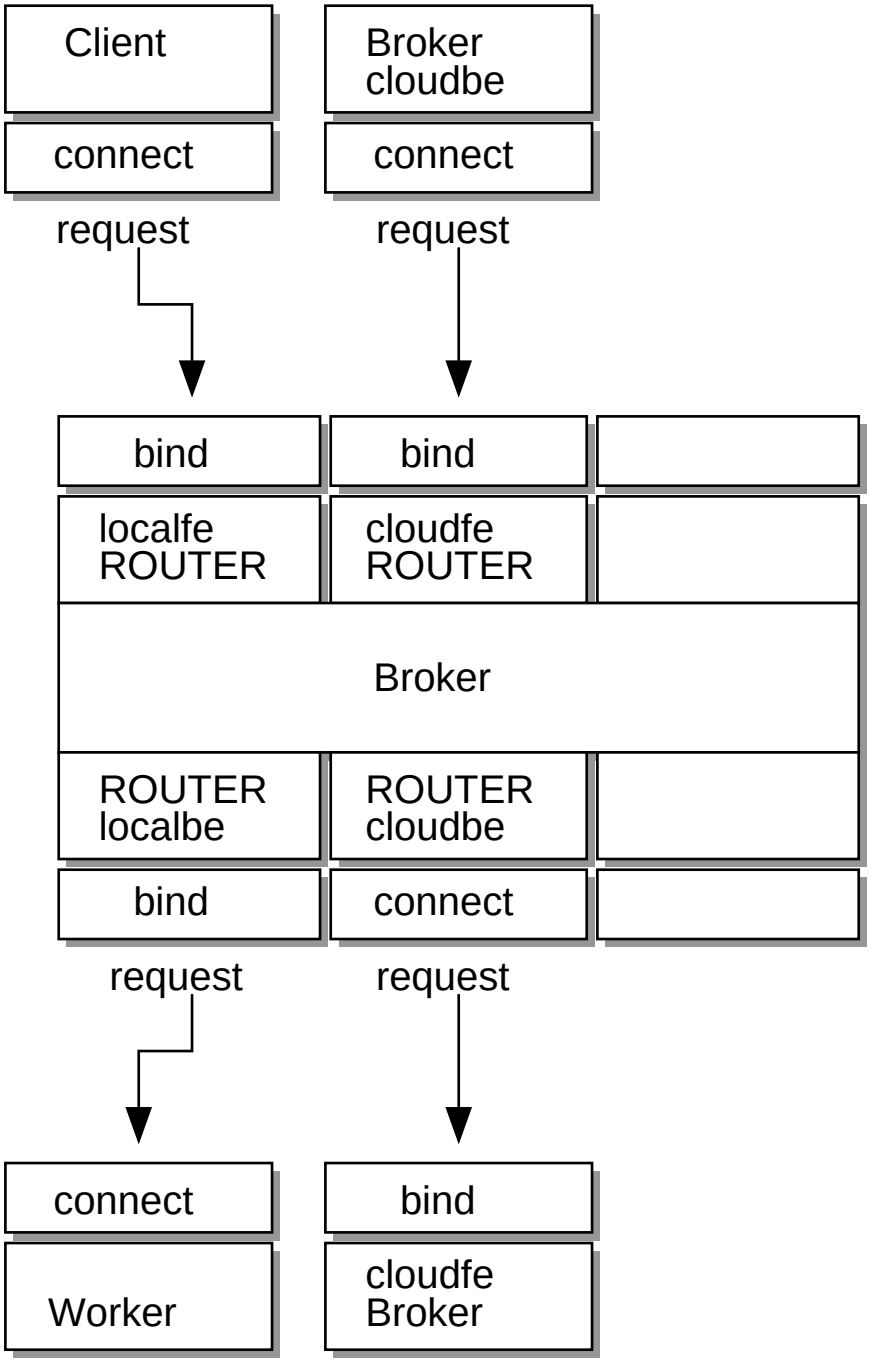
In real life we'd not send out state messages at regular intervals but rather whenever we had a state change, i.e. whenever a worker becomes available or unavailable. That may seem like a lot of traffic but state messages are small and we've established that the inter-cluster connections are super-fast.

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.

3.9.7. Prototyping the Local and Cloud Flows

Let's now prototype at the flow of tasks via the local and cloud sockets(Figure 3-27). This code pulls requests from clients and then distributes them to local workers and cloud peers on a random basis.

Figure 3-27. 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 dumb 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".

Example 3-10. Prototype local and cloud flow (peering2.lua)

```
--
--  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.
--
--  Author: Robert G. Jakabosky <bobby@sharedrealm.com>
--

require"zmq"
require"zmq.poller"
```

```
require"zmq.threads"
require"zmsg"

local tremove = table.remove

local NBR_CLIENTS  = 10
local NBR_WORKERS  = 3

local pre_code = [[
    local self, seed = ...
    local zmq = require"zmq"
    local zmsg = require"zmsg"
    require"zhelpers"
    math.randomseed(seed)
    local context = zmq.init(1)

]]

--  Request-reply client using REQ socket
--
local client_task = pre_code .. [[
    local client = context:socket(zmq.REQ)
    local endpoint = string.format("ipc://%s-localfe.ipc", self)
    assert(client:connect(endpoint))

    while true do
        --  Send request, get reply
        local msg = zmsg.new ("HELLO")
        msg:send(client)
        msg = zmsg.recv (client)
        printf ("I: client status: %s\n", msg:body())
    end
    --  We never get here but if we did, this is how we'd exit cleanly
    client:close()
    context:term()
]]

--  Worker using REQ socket to do LRU routing
--
local worker_task = pre_code .. [[
    local worker = context:socket(zmq.REQ)
    local endpoint = string.format("ipc://%s-localbe.ipc", self)
    assert(worker:connect(endpoint))

    --  Tell broker we're ready for work
    local msg = zmsg.new ("READY")
    msg:send(worker)

    while true do
        msg = zmsg.recv (worker)
        --  Do some 'work'
        s_sleep (1000)
        msg:body_fmt("OK - %04x", randof (0x10000))
```

```
        msg:send(worker)
    end
    --  We never get here but if we did, this is how we'd exit cleanly
    worker:close()
    context:term()
]]

--  First argument is this broker's name
--  Other arguments are our peers' names
--
s_version_assert (2, 1)
if (#arg < 1) then
    printf ("syntax: peering2 me doyouend...\n")
    os.exit(-1)
end
--  Our own name; in practice this'd be configured per node
local self = arg[1]
printf ("I: preparing broker at %s...\n", self)
math.randomseed(os.time())

--  Prepare our context and sockets
local context = zmq.init(1)

--  Bind cloud frontend to endpoint
local cloudfe = context:socket(zmq.ROUTER)
local endpoint = string.format("ipc://%s-cloud.ipc", self)
cloudfe:setopt(zmq.IDENTITY, self)
assert(cloudfe:bind(endpoint))

--  Connect cloud backend to all peers
local cloudbe = context:socket(zmq.ROUTER)
cloudbe:setopt(zmq.IDENTITY, self)

local peers = {}
for n=2,#arg do
    local peer = arg[n]
    -- add peer name to peers list.
    peers[#peers + 1] = peer
    peers[peer] = true -- map peer's name to 'true' for fast lookup
    printf ("I: connecting to cloud frontend at '%s'\n", peer)
    local endpoint = string.format("ipc://%s-cloud.ipc", peer)
    assert(cloudbe:connect(endpoint))
end
--  Prepare local frontend and backend
local localfe = context:socket(zmq.ROUTER)
local endpoint = string.format("ipc://%s-localfe.ipc", self)
assert(localfe:bind(endpoint))

local localbe = context:socket(zmq.ROUTER)
local endpoint = string.format("ipc://%s-localbe.ipc", self)
assert(localbe:bind(endpoint))

--  Get user to tell us when we can start...
```

```
printf ("Press Enter when all brokers are started: ")
io.read('*l')

--  Start local workers
local workers = {}
for n=1,NBR_WORKERS do
    local seed = os.time() + math.random()
    workers[n] = zmq.threads.runstring(nil, worker_task, self, seed)
    workers[n]:start(true)
end
--  Start local clients
local clients = {}
for n=1,NBR_CLIENTS do
    local seed = os.time() + math.random()
    clients[n] = zmq.threads.runstring(nil, client_task, self, seed)
    clients[n]:start(true)
end

--  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
local worker_queue = {}
local backends = zmq.poller(2)

local function send_reply(msg)
    local address = msg:address()
    -- Route reply to cloud if it's addressed to a broker
    if peers[address] then
        msg:send(cloudfe) -- reply is for a peer.
    else
        msg:send(localfe) -- reply is for a local client.
    end
end

backends:add(localbe, zmq.POLLIN, function()
    local msg = zmsg.recv(localbe)

    --  Use worker address for LRU routing
    worker_queue[#worker_queue + 1] = msg:unwrap()
    -- if reply is not "READY" then route reply back to client.
    if (msg:address() ~= "READY") then
        send_reply(msg)
    end
end)

backends:add(cloudbe, zmq.POLLIN, function()
    local msg = zmsg.recv(cloudbe)
    --  We don't use peer broker address for anything
    msg:unwrap()
```

```
    -- send reply back to client.
    send_reply(msg)
end)

local frontends = zmq.poller(2)
local localfe_ready = false
local cloudfe_ready = false

frontends:add(localfe, zmq.POLLIN, function() localfe_ready = true end)
frontends:add(cloudfe, zmq.POLLIN, function() cloudfe_ready = true end)

while true do
    local timeout = (#worker_queue > 0) and 1000000 or -1
    --  If we have no workers anyhow, wait indefinitely
    rc = backends:poll(timeout)
    assert (rc >= 0)

    --  Now route as many clients requests as we can handle
    --
    while (#worker_queue > 0) do
        rc = frontends:poll(0)
        assert (rc >= 0)
        local reroutable = false
        local msg
        --  We'll do peer brokers first, to prevent starvation
        if (cloudfe_ready) then
            cloudfe_ready = false -- reset flag
            msg = zmsg.recv (cloudfe)
            reroutable = false
        elseif (localfe_ready) then
            localfe_ready = false -- reset flag
            msg = zmsg.recv (localfe)
            reroutable = true
        else
            break;      --  No work, go back to backends
        end

        --  If reroutable, send to cloud 20% of the time
        --  Here we'd normally use cloud status information
        --
        local percent = randof (5)
        if (reroutable and #peers > 0 and percent == 0) then
            --  Route to random broker peer
            local random_peer = randof (#peers) + 1
            msg:wrap(peers[random_peer], nil)
            msg:send(cloudbe)
        else
            --  Dequeue and drop the next worker address
            local worker = tremove(worker_queue, 1)
            msg:wrap(worker, "")
            msg:send(localbe)
        end
    end
```

```
end
--  We never get here but clean up anyhow
localbe:close()
cloudbe:close()
localfe:close()
cloudfe:close()
context:term()
```

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 an abstraction that works. If you build ØMQ applications in C, you should use CZMQ.
- 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.

3.9.8. Putting it All Together

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:

Example 3-11. Full cluster simulation (peering3.lua)

```
--
--  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.
--
--  Author: Robert G. Jakabosky <bobby@sharedrealm.com>
```

```
--

require"zmq"
require"zmq.poller"
require"zmq.threads"
require"zmsg"

local tremove = table.remove

local NBR_CLIENTS  = 10
local NBR_WORKERS  = 5

local pre_code = [[
    local self, seed = ...
    local zmq = require"zmq"
    local zmsg = require"zmsg"
    require"zhelpers"
    math.randomseed(seed)
    local context = zmq.init(1)

]]

--  Request-reply client using REQ socket
--  To simulate load, clients issue a burst of requests and then
--  sleep for a random period.
--
local client_task = pre_code .. [[
    require"zmq.poller"

    local client = context:socket(zmq.REQ)
    local endpoint = string.format("ipc://%s-localfe.ipc", self)
    assert(client:connect(endpoint))

    local monitor = context:socket(zmq.PUSH)
    local endpoint = string.format("ipc://%s-monitor.ipc", self)
    assert(monitor:connect(endpoint))

    local poller = zmq.poller(1)
    local task_id = nil

    poller:add(client, zmq.POLLIN, function()
        local msg = zmsg.recv (client)
        --  Worker is supposed to answer us with our task id
        assert (msg:body() == task_id)
        -- mark task as processed.
        task_id = nil
    end)
    local is_running = true
    while is_running do
        s_sleep (randof (5) * 1000)

        local burst = randof (15)
        while (burst > 0) do
```

```
            burst = burst - 1
            --  Send request with random hex ID
            task_id = string.format("%04X", randof (0x10000))
            local msg = zmsg.new(task_id)
            msg:send(client)

            --  Wait max ten seconds for a reply, then complain
            rc = poller:poll(10 * 1000000)
            assert (rc >= 0)

            if task_id then
                local msg = zmsg.new()
                msg:body_fmt(
                    "E: CLIENT EXIT - lost task %s", task_id)
                msg:send(monitor)
                -- exit event loop
                is_running = false
                break
            end
        end
    end
    --  We never get here but if we did, this is how we'd exit cleanly
    client:close()
    monitor:close()
    context:term()
]]

--  Worker using REQ socket to do LRU routing
--
local worker_task = pre_code .. [[
    local worker = context:socket(zmq.REQ)
    local endpoint = string.format("ipc://%s-localbe.ipc", self)
    assert(worker:connect(endpoint))

    --  Tell broker we're ready for work
    local msg = zmsg.new ("READY")
    msg:send(worker)

    while true do
        --  Workers are busy for 0/1/2 seconds
        msg = zmsg.recv (worker)
        s_sleep (randof (2) * 1000)
        msg:send(worker)
    end
    --  We never get here but if we did, this is how we'd exit cleanly
    worker:close()
    context:term()
]]

--  First argument is this broker's name
--  Other arguments are our peers' names
--
s_version_assert (2, 1)
```

```
if (#arg < 1) then
    printf ("syntax: peering3 me doyouend...\n")
    os.exit(-1)
end
--  Our own name; in practice this'd be configured per node
local self = arg[1]
printf ("I: preparing broker at %s...\n", self)
math.randomseed(os.time())

--  Prepare our context and sockets
local context = zmq.init(1)

--  Bind cloud frontend to endpoint
local cloudfe = context:socket(zmq.ROUTER)
local endpoint = string.format("ipc://%s-cloud.ipc", self)
cloudfe:setopt(zmq.IDENTITY, self)
assert(cloudfe:bind(endpoint))

--  Bind state backend / publisher to endpoint
local statebe = context:socket(zmq.PUB)
local endpoint = string.format("ipc://%s-state.ipc", self)
assert(statebe:bind(endpoint))

--  Connect cloud backend to all peers
local cloudbe = context:socket(zmq.ROUTER)
cloudbe:setopt(zmq.IDENTITY, self)

for n=2,#arg do
    local peer = arg[n]
    printf ("I: connecting to cloud frontend at '%s'\n", peer)
    local endpoint = string.format("ipc://%s-cloud.ipc", peer)
    assert(cloudbe:connect(endpoint))
end
--  Connect statefe to all peers
local statefe = context:socket(zmq.SUB)
statefe:setopt(zmq.SUBSCRIBE, "", 0)

local peers = {}
for n=2,#arg do
    local peer = arg[n]
    -- add peer name to peers list.
    peers[#peers + 1] = peer
    peers[peer] = 0 -- set peer's initial capacity to zero.
    printf ("I: connecting to state backend at '%s'\n", peer)
    local endpoint = string.format("ipc://%s-state.ipc", peer)
    assert(statefe:connect(endpoint))
end
--  Prepare local frontend and backend
local localfe = context:socket(zmq.ROUTER)
local endpoint = string.format("ipc://%s-localfe.ipc", self)
assert(localfe:bind(endpoint))

local localbe = context:socket(zmq.ROUTER)
```

```
local endpoint = string.format("ipc://%s-localbe.ipc", self)
assert(localbe:bind(endpoint))

--  Prepare monitor socket
local monitor = context:socket(zmq.PULL)
local endpoint = string.format("ipc://%s-monitor.ipc", self)
assert(monitor:bind(endpoint))

--  Start local workers
local workers = {}
for n=1,NBR_WORKERS do
    local seed = os.time() + math.random()
    workers[n] = zmq.threads.runstring(nil, worker_task, self, seed)
    workers[n]:start(true)
end
--  Start local clients
local clients = {}
for n=1,NBR_CLIENTS do
    local seed = os.time() + math.random()
    clients[n] = zmq.threads.runstring(nil, client_task, self, seed)
    clients[n]:start(true)
end

--  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
local local_capacity = 0
local cloud_capacity = 0
local worker_queue = {}
local backends = zmq.poller(2)

local function send_reply(msg)
    local address = msg:address()
    -- Route reply to cloud if it’s addressed to a broker
    if peers[address] then
        msg:send(cloudfe) -- reply is for a peer.
    else
        msg:send(localfe) -- reply is for a local client.
    end
end

backends:add(localbe, zmq.POLLIN, function()
    local msg = zmsg.recv(localbe)

    --  Use worker address for LRU routing
    local_capacity = local_capacity + 1
```

```
    worker_queue[local_capacity] = msg:unwrap()
    -- if reply is not "READY" then route reply back to client.
    if (msg:address() ~= "READY") then
        send_reply(msg)
    end
end)

backends:add(cloudbe, zmq.POLLIN, function()
    local msg = zmsg.recv(cloudbe)

    --  We don't use peer broker address for anything
    msg:unwrap()
    -- send reply back to client.
    send_reply(msg)
end)

backends:add(statefe, zmq.POLLIN, function()
    local msg = zmsg.recv (statefe)
    -- TODO: track capacity for each peer
    cloud_capacity = tonumber(msg:body())
end)

backends:add(monitor, zmq.POLLIN, function()
    local msg = zmsg.recv (monitor)
    printf("%s\n", msg:body())
end)

local frontends = zmq.poller(2)
local localfe_ready = false
local cloudfe_ready = false

frontends:add(localfe, zmq.POLLIN, function() localfe_ready = true end)
frontends:add(cloudfe, zmq.POLLIN, function() cloudfe_ready = true end)

local MAX_BACKEND_REPLIES = 20

while true do
    -- If we have no workers anyhow, wait indefinitely
    local timeout = (local_capacity > 0) and 1000000 or -1
    local rc, err = backends:poll(timeout)
    assert (rc >= 0, err)

    --  Track if capacity changes during this iteration
    local previous = local_capacity

    --  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) > 0) do
        local rc, err = frontends:poll(0)
        assert (rc >= 0, err)
```

```
        if (localfe_ready) then
            localfe_ready = false
            msg = zmsg.recv (localfe)
        elseif (cloudfe_ready and local_capacity > 0) then
            cloudfe_ready = false
            -- we have local capacity poll cloud frontend for work.
            msg = zmsg.recv (cloudfe)
        else
            break;      --  No work, go back to primary
        end

        if (local_capacity > 0) then
            --  Dequeue and drop the next worker address
            local worker = tremove(worker_queue, 1)
            local_capacity = local_capacity - 1
            msg:wrap(worker, "")
            msg:send(localbe)
        else
            --  Route to random broker peer
            printf ("I: route request %s to cloud...\n",
                msg:body())
            local random_peer = randof (#peers) + 1
            msg:wrap(peers[random_peer], nil)
            msg:send(cloudbe)
        end
    end
    if (local_capacity ~= previous) then
        --  Broadcast new capacity
        local msg = zmsg.new()
        -- TODO: send our name with capacity.
        msg:body_fmt("%d", local_capacity)
        --  We stick our own address onto the envelope
        msg:wrap(self, nil)
        msg:send(statebe)
    end
end
--  We never get here but clean up anyhow
localbe:close()
cloudbe:close()
localfe:close()
cloudfe:close()
statefe:close()
monitor:close()
context:term()
```

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 4. Reliable Request-Reply

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 request-reply 'patterns', reusable models that help you design your own ØMQ architectures:

- 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 fail-over.
- The *Freelance* pattern: brokerless reliable request-reply.

4.1. What is "Reliability"?

Most people who speak of 'reliability' don't really know what they mean. We can only define reliability in terms of failure. That is, if we can handle a certain set of well-defined and understood 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 for the same reasons as application code. System code *should* be more reliable than application code but it can still crash and burn, and especially run out of memory if it tries to queue messages 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. So we get "lost" messages.
- Networks can fail (e.g. wifi gets switched off or goes out of range). ØMQ will automatically reconnect in such cases but in the meantime, messages may get lost.
- 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, which also told me that 78% of statistics are made up on the spot), that's what we'll look at. If you're a large company with money to spend on the last two cases, contact my company immediately! There's a large hole behind my beach house waiting to be converted into an executive pool.

4.2. Designing Reliability

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 dies, 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 just on request-reply, which is the low-hanging Durian fruit of reliable messaging. We'll cover reliable pub-sub and pipeline in later 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 much 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 case where you can really

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 (RRR) patterns I like to call the "Pirate" patterns (you'll get the joke, eventually).

There are in my experience, roughly three ways to connect clients to servers. Each needs 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.

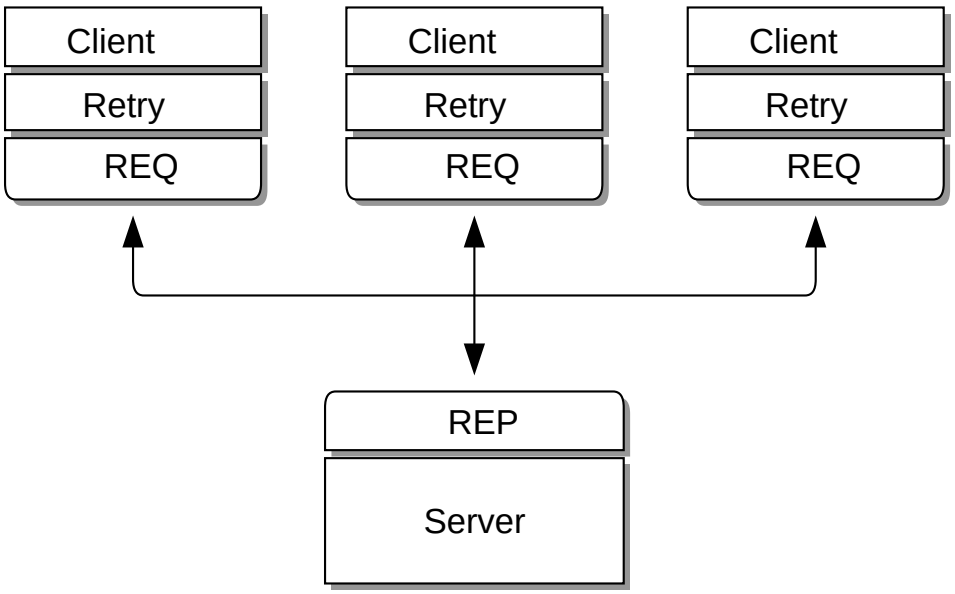
Each of these has their trade-offs and often you'll mix them. We'll look at all three of these in detail.

4.3. Client-side Reliability (Lazy Pirate Pattern)

We can get very simple reliable request-reply with only some changes in the client. We call this the Lazy Pirate pattern(Figure 4-1). 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, if no reply arrived within a timeout period.
- Abandon the transaction if after several requests, there is still no reply.

Figure 4-1. The 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:

Example 4-1. Lazy Pirate client (lpclient.lua)

```
--
--  Lazy Pirate client
--  Use zmq_poll to do a safe request-reply
--  To run, start lpserver and then randomly kill/restart it
--
--  Author: Robert G. Jakabosky <bobby@sharedrealm.com>
--

require"zmq"
require"zmq.poller"
require"zhelpers"

local REQUEST_TIMEOUT      = 2500    --  msecs, (> 1000!)
local REQUEST_RETRIES      = 3       --  Before we abandon

--  Helper function that returns a new configured socket
--  connected to the Hello World server
```

```
--
local function s_client_socket(context)
    printf ("I: connecting to server...\n")
    local client = context:socket(zmq.REQ)
    client:connect("tcp://localhost:5555")

    --  Configure socket to not wait at close time
    client:setopt(zmq.LINGER, 0)
    return client
end
s_version_assert (2, 1)
local context = zmq.init(1)
local client = s_client_socket (context)

local sequence = 0
local retries_left = REQUEST_RETRIES
local expect_reply = true

local poller = zmq.poller(1)

local function client_cb()
    --  We got a reply from the server, must match sequence
    --local reply = assert(client:recv(zmq.NOBLOCK))
    local reply = client:recv()
    if (tonumber(reply) == sequence) then
        printf ("I: server replied OK (%s)\n", reply)
        retries_left = REQUEST_RETRIES
        expect_reply = false
    else
        printf ("E: malformed reply from server: %s\n", reply)
    end
end
poller:add(client, zmq.POLLIN, client_cb)

while (retries_left > 0) do
    sequence = sequence + 1
    --  We send a request, then we work to get a reply
    local request = string.format("%d", sequence)
    client:send(request)
    expect_reply = true

    while (expect_reply) do
        --  Poll socket for a reply, with timeout
        local cnt = assert(poller:poll(REQUEST_TIMEOUT * 1000))

        -- Check if there was no reply
        if (cnt == 0) then
            retries_left = retries_left - 1
            if (retries_left == 0) then
                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
                poller:remove(client)
                client:close()
                client = s_client_socket (context)
                poller:add(client, zmq.POLLIN, client_cb)
                --  Send request again, on new socket
                client:send(request)
            end
        end
    end
end
client:close()
context:term()
```

Run this together with the matching server:

Example 4-2. Lazy Pirate server (lpserver.lua)

```
--
--  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.
--
--  Author: Robert G. Jakabosky <bobby@sharedrealm.com>
--
require"zmq"
require"zhelpers"

math.randomseed(os.time())

local context = zmq.init(1)
local server = context:socket(zmq.REP)
server:bind("tcp://*:5555")

local cycles = 0
while true do
    local request = server:recv()
    cycles = cycles + 1

    --  Simulate various problems, after a few cycles
    if (cycles > 3 and randof (3) == 0) then
        printf("I: simulating a crash\n")
        break
    elseif (cycles > 3 and randof (3) == 0) then
        printf("I: simulating CPU overload\n")
        s_sleep(2000)
    end
    printf("I: normal request (%s)\n", request)
    s_sleep(1000)              --  Do some heavy work
    server:send(request)
```

```
end
server:close()
context:term()
```

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 that 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 fail-over to backup / alternate servers.

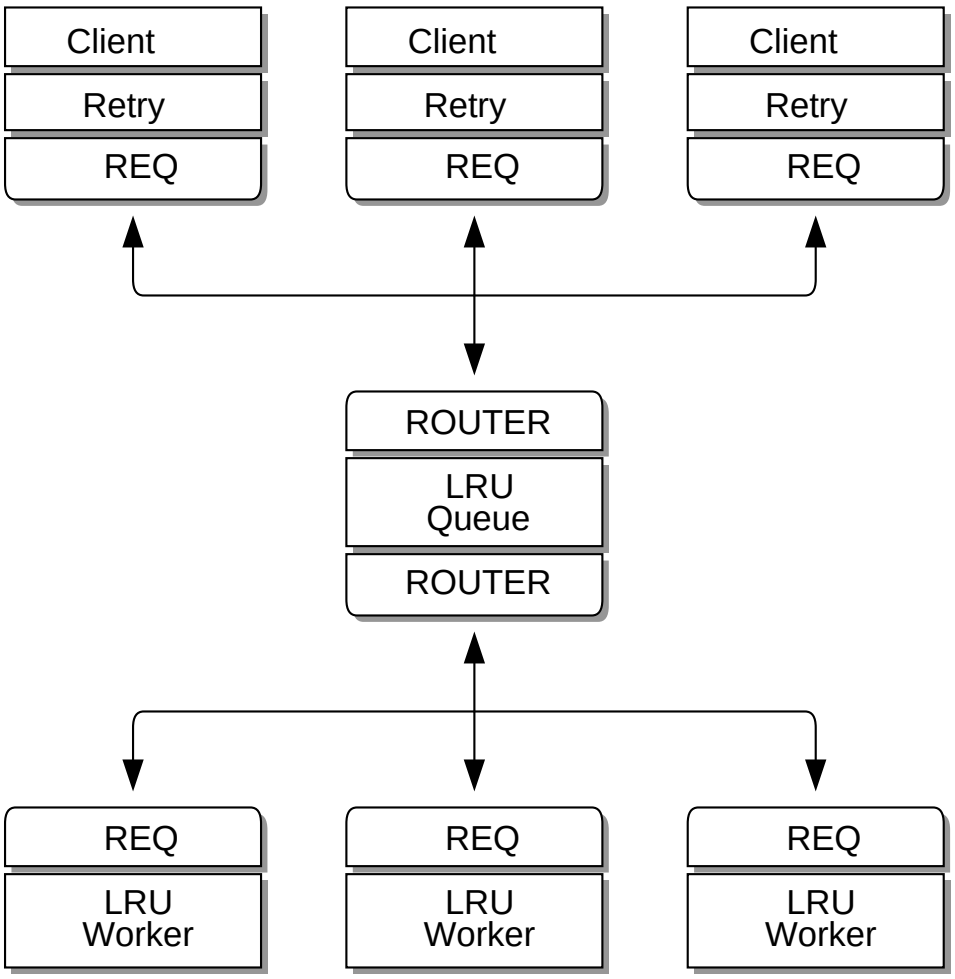
4.4. Basic Reliable Queuing (Simple Pirate Pattern)

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, it's 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 4-2).

Figure 4-2. The 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:

Example 4-3. Simple Pirate queue (spqueue.lua)

```
--
--  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.
--
--  Author: Robert G. Jakabosky <bobby@sharedrealm.com>
--
```

```
require"zmq"
require"zmq.poller"
require"zhelpers"
require"zmsg"

local tremove = table.remove

local MAX_WORKERS  = 100

s_version_assert (2, 1)

--  Prepare our context and sockets
local context = zmq.init(1)
local frontend = context:socket(zmq.ROUTER)
local backend  = context:socket(zmq.ROUTER)
frontend:bind("tcp://*:5555");    --  For clients
backend:bind("tcp://*:5556");    --  For workers

--  Queue of available workers
local worker_queue = {}
local is_accepting = false

local poller = zmq.poller(2)

local function frontend_cb()
    --  Now get next client request, route to next worker
    local msg = zmsg.recv (frontend)

    -- Dequeue a worker from the queue.
    local worker = tremove(worker_queue, 1)

    msg:wrap(worker, "")
    msg:send(backend)

    if (#worker_queue == 0) then
        -- stop accepting work from clients, when no workers are available.
        poller:remove(frontend)
        is_accepting = false
    end
end

--  Handle worker activity on backend
poller:add(backend, zmq.POLLIN, function()
    local msg = zmsg.recv(backend)
    --  Use worker address for LRU routing
    worker_queue[#worker_queue + 1] = msg:unwrap()

    -- start accepting client requests, if we are not already doing so.
    if not is_accepting then
        is_accepting = true
        poller:add(frontend, zmq.POLLIN, frontend_cb)
    end
```

```
    --  Forward message to client if it's not a READY
    if (msg:address() ~= "READY") then
        msg:send(frontend)
    end
end)

-- start poller's event loop
poller:start()

--  We never exit the main loop
```

Here is the worker, which takes the Lazy Pirate server and adapts it for the LRU pattern (using the REQ 'ready' signaling):

Example 4-4. Simple Pirate worker (spworker.lua)

```
--
--  Simple Pirate worker
--  Connects REQ socket to tcp://*:5556
--  Implements worker part of LRU queueing
--
--  Author: Robert G. Jakabosky <bobby@sharedrealm.com>
--
require"zmq"
require"zmsg"

math.randomseed(os.time())

local context = zmq.init(1)
local worker = context:socket(zmq.REQ)

--  Set random identity to make tracing easier
local identity = string.format("%04X-%04X", randof (0x10000), randof (0x10000))
worker:setopt(zmq.IDENTITY, identity)
worker:connect("tcp://localhost:5556")

--  Tell queue we're ready for work
printf ("I: (%s) worker ready\n", identity)
worker:send("READY")

local cycles = 0
while true do
    local msg = zmsg.recv (worker)

    --  Simulate various problems, after a few cycles
    cycles = cycles + 1
    if (cycles > 3 and randof (5) == 0) then
        printf ("I: (%s) simulating a crash\n", identity)
        break
    elseif (cycles > 3 and randof (5) == 0) then
        printf ("I: (%s) simulating CPU overload\n", identity)
        s_sleep (5000)
```

```
    end
    printf ("I: (%s) normal reply - %s\n",
            identity, msg:body())
    s_sleep (1000)              --  Do some heavy work
    msg:send(worker)
end
worker:close()
context:term()
```

To test this, start a handful 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.

4.5. Robust Reliable Queuing (Paranoid Pirate Pattern)

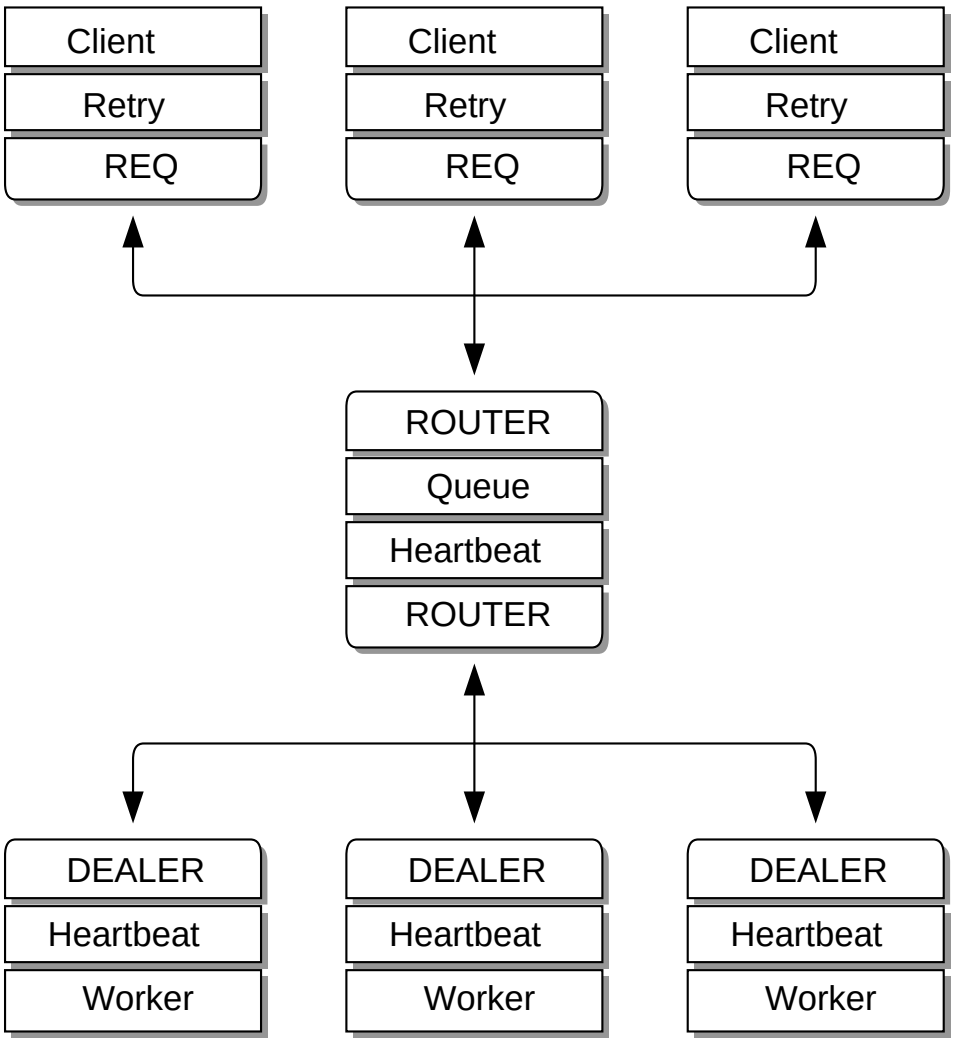
The Simple Pirate Queue pattern works pretty well, especially since it's just a combination of two existing patterns, but it has some weaknesses:

- 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 signaled "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(Figure 4-3). 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 4-3. The Paranoid Pirate Pattern



We're still using the Lazy Pirate client. Here is the Paranoid Pirate queue device:

Example 4-5. Paranoid Pirate queue (ppqueue.lua)

```
--
--  Paranoid Pirate queue
--
--  Author: Robert G. Jakabosky <bobby@sharedrealm.com>
--
require"zmq"
require"zmq.poller"
```

```
require"zmsg"

local MAX_WORKERS          = 100
local HEARTBEAT_LIVENESS   = 3       --  3-5 is reasonable
local HEARTBEAT_INTERVAL   = 1000    --  msecs

local tremove = table.remove

--  Insert worker at end of queue, reset expiry
--  Worker must not already be in queue
local function s_worker_append(queue, identity)
    if queue[identity] then
        printf ("E: duplicate worker identity %s", identity)
    else
        assert (#queue < MAX_WORKERS)
        queue[identity] = s_clock() + HEARTBEAT_INTERVAL * HEARTBEAT_LIVENESS
        queue[#queue + 1] = identity
    end
end
--  Remove worker from queue, if present
local function s_worker_delete(queue, identity)
    for i=1,#queue do
        if queue[i] == identity then
            tremove(queue, i)
            break
        end
    end
    queue[identity] = nil
end
--  Reset worker expiry, worker must be present
local function s_worker_refresh(queue, identity)
    if queue[identity] then
        queue[identity] = s_clock() + HEARTBEAT_INTERVAL * HEARTBEAT_LIVENESS
    else
        printf("E: worker %s not ready\n", identity)
    end
end
--  Pop next available worker off queue, return identity
local function s_worker_dequeue(queue)
    assert (#queue > 0)
    local identity = tremove(queue, 1)
    queue[identity] = nil
    return identity
end
--  Look for & kill expired workers
local function s_queue_purge(queue)
    local curr_clock = s_clock()
    --  Work backwards from end to simplify removal
    for i=#queue,1,-1 do
        local id = queue[i]
        if (curr_clock > queue[id]) then
            tremove(queue, i)
            queue[id] = nil
```

```
        end
    end
end
s_version_assert (2, 1)

--  Prepare our context and sockets
local context = zmq.init(1)
local frontend = context:socket(zmq.ROUTER)
local backend  = context:socket(zmq.ROUTER)
frontend:bind("tcp://*:5555");    --  For clients
backend:bind("tcp://*:5556");    --  For workers

--  Queue of available workers
local queue = {}
local is_accepting = false

--  Send out heartbeats at regular intervals
local heartbeat_at = s_clock() + HEARTBEAT_INTERVAL

local poller = zmq.poller(2)

local function frontend_cb()
    --  Now get next client request, route to next worker
    local msg = zmsg.recv(frontend)
    local identity = s_worker_dequeue (queue)
    msg:push(identity)
    msg:send(backend)

    if (#queue == 0) then
        -- stop accepting work from clients, when no workers are available.
        poller:remove(frontend)
        is_accepting = false
    end
end

--  Handle worker activity on backend
poller:add(backend, zmq.POLLIN, function()
    local msg = zmsg.recv(backend)
    local identity = msg:unwrap()

    --  Return reply to client if it’s not a control message
    if (msg:parts() == 1) then
        if (msg:address() == "READY") then
            s_worker_delete(queue, identity)
            s_worker_append(queue, identity)
        elseif (msg:address() == "HEARTBEAT") then
            s_worker_refresh(queue, identity)
        else
            printf("E: invalid message from %s\n", identity)
            msg:dump()
        end
    else
        -- reply for client.
```

```
        msg:send(frontend)
        s_worker_append(queue, identity)
    end

    -- start accepting client requests, if we are not already doing so.
    if not is_accepting and #queue > 0 then
        is_accepting = true
        poller:add(frontend, zmq.POLLIN, frontend_cb)
    end
end)

-- start poller's event loop
while true do
    local cnt = assert(poller:poll(HEARTBEAT_INTERVAL * 1000))
    --  Send heartbeats to idle workers if it's time
    if (s_clock() > heartbeat_at) then
        for i=1,#queue do
            local msg = zmsg.new("HEARTBEAT")
            msg:wrap(queue[i], nil)
            msg:send(backend)
        end
        heartbeat_at = s_clock() + HEARTBEAT_INTERVAL
    end
    s_queue_purge(queue)
end

--  We never exit the main loop
--  But pretend to do the right shutdown anyhow
while (#queue > 0) do
    s_worker_dequeue(queue)
end

frontend:close()
backend:close()
```

The queue extends the LRU pattern with heartbeating of workers. Heartbeating is one of those 'simple' things that can be subtle to get right. I'll explain more about that in a second.

Here is the Paranoid Pirate worker:

Example 4-6. Paranoid Pirate worker (ppworker.lua)

```
--
--  Paranoid Pirate worker
--
--  Author: Robert G. Jakabosky <bobby@sharedrealm.com>
--
require"zmq"
require"zmq.poller"
require"zmsg"
```

```
local HEARTBEAT_LIVENESS   = 3       --  3-5 is reasonable
local HEARTBEAT_INTERVAL   = 1000    --  msecs
local INTERVAL_INIT        = 1000    --  Initial reconnect
local INTERVAL_MAX        = 32000    --  After exponential backoff

--  Helper function that returns a new configured socket
--  connected to the Hello World server
--
local identity

local function s_worker_socket (context)
    local worker = context:socket(zmq.DEALER)

    --  Set random identity to make tracing easier
    identity = string.format("%04X-%04X", randof (0x10000), randof (0x10000))
    worker:setopt(zmq.IDENTITY, identity)
    worker:connect("tcp://localhost:5556")

    --  Configure socket to not wait at close time
    worker:setopt(zmq.LINGER, 0)

    --  Tell queue we're ready for work
    printf("I: (%s) worker ready\n", identity)
    worker:send("READY")

    return worker
end

s_version_assert (2, 1)
math.randomseed(os.time())

local context = zmq.init(1)
local worker = s_worker_socket (context)

--  If liveness hits zero, queue is considered disconnected
local liveness = HEARTBEAT_LIVENESS
local interval = INTERVAL_INIT

--  Send out heartbeats at regular intervals
local heartbeat_at = s_clock () + HEARTBEAT_INTERVAL

local poller = zmq.poller(1)

local is_running = true

local cycles = 0
local function worker_cb()
    --  Get message
    --  - 3-part envelope + content -> request
    --  - 1-part "HEARTBEAT" -> heartbeat
    local msg = zmsg.recv (worker)

    if (msg:parts() == 3) then
```

```
        --  Simulate various problems, after a few cycles
        cycles = cycles + 1
        if (cycles > 3 and randof (5) == 0) then
            printf ("I: (%s) simulating a crash\n", identity)
            is_running = false
            return
        elseif (cycles > 3 and randof (5) == 0) then
            printf ("I: (%s) simulating CPU overload\n",
                identity)
            s_sleep (5000)
        end
        printf ("I: (%s) normal reply - %s\n",
            identity, msg:body())
        msg:send(worker)
        liveness = HEARTBEAT_LIVENESS
        s_sleep(1000);           --  Do some heavy work
    elseif (msg:parts() == 1 and msg:body() == "HEARTBEAT") then
        liveness = HEARTBEAT_LIVENESS
    else
        printf ("E: (%s) invalid message\n", identity)
        msg:dump()
    end
    interval = INTERVAL_INIT
end
poller:add(worker, zmq.POLLIN, worker_cb)

while is_running do
    local cnt = assert(poller:poll(HEARTBEAT_INTERVAL * 1000))

    if (cnt == 0) then
        liveness = liveness - 1
        if (liveness == 0) then
            printf ("W: (%s) heartbeat failure, can't reach queue\n",
                identity)
            printf ("W: (%s) reconnecting in %d msec...\n",
                identity, interval)
            s_sleep (interval)

            if (interval < INTERVAL_MAX) then
                interval = interval * 2
            end
            poller:remove(worker)
            worker:close()
            worker = s_worker_socket (context)
            poller:add(worker, zmq.POLLIN, worker_cb)
            liveness = HEARTBEAT_LIVENESS
        end
    end
    --  Send heartbeat to queue if it's time
    if (s_clock () > heartbeat_at) then
        heartbeat_at = s_clock () + HEARTBEAT_INTERVAL
        printf("I: (%s) worker heartbeat\n", identity)
        worker:send("HEARTBEAT")
```

```
    end
end
worker:close()
context:term()
```

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.
- 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 no 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.

4.6. Heartbeating

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.
- 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 you'll enter the loop every time you receive any message (fun, when you have two peers sending each other heartbeats) (think about it).

Your language or binding should provide a 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 = zclock_time () + HEARTBEAT_INTERVAL;

while (1) {
    ...
    int rc = zmq_poll (items, 1, HEARTBEAT_INTERVAL * ZMQ_POLL_MSEC);
    ...
    //  Send heartbeat to queue if it's time
    if (zclock_time () > heartbeat_at) {
        ... Send heartbeats to all peers that expect them
        //  Set timer for next heartbeat
        heartbeat_at = zclock_time () + 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 (shortest time) 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. It does make things more complex, but we do that in real applications so that clients can detect when brokers die, and do clever things like switch to alternate brokers.

4.7. Contracts and Protocols

If you're paying attention you'll realize that Paranoid Pirate is not interoperable with Simple Pirate, because of the heartbeats. But how do we define "interoperable"? To guarantee interoperability we need a kind of contract, an agreement that lets different teams, in different times and places, write code that is guaranteed to work together. We call this a "protocol".

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 (http://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 (http://rfc.zeromq.org/spec:6). 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.

4.8. Service-Oriented Reliable Queuing (Majordomo Pattern)

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:

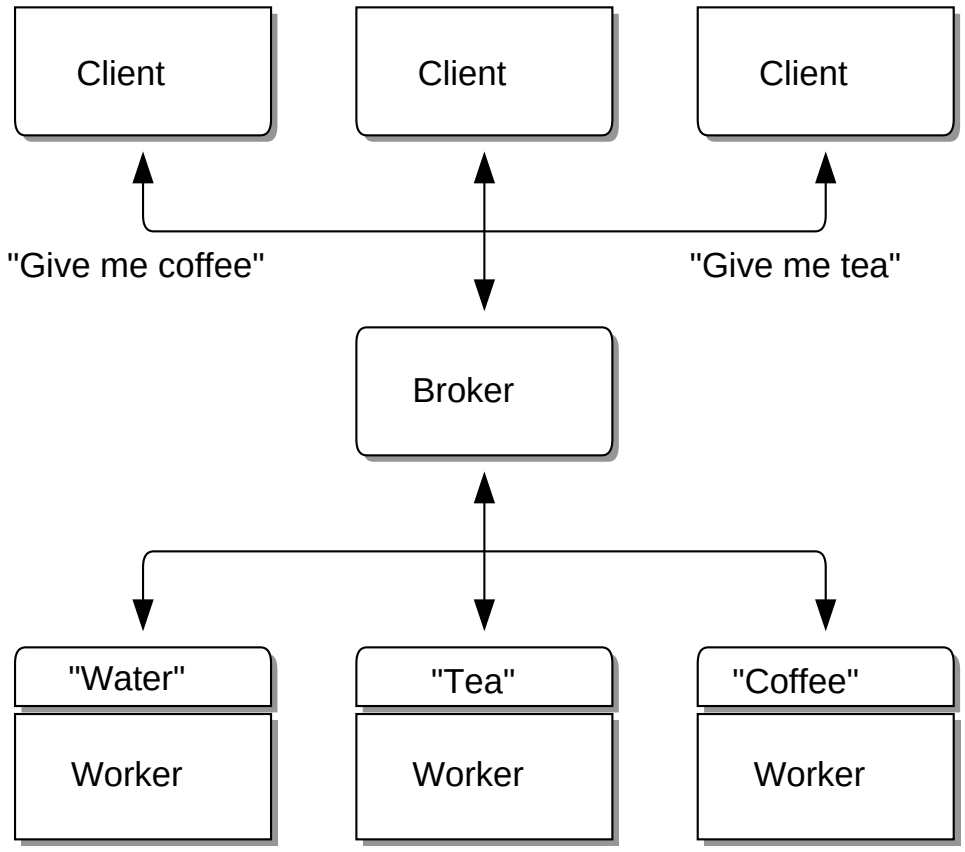
- http://rfc.zeromq.org/spec:7

This one-page specification takes PPP and turns it into something more solid(Figure 4-4). 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 4-4. The 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 CZMQ binding (http://czmq.zeromq.org/):

```
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:

Example 4-7. Majordomo client API (mdcliapi.lua)

```
--
-- mdcliapi.lua - Majordomo Protocol Client API
--
-- Author: Robert G. Jakabosky <bobby@sharedrealm.com>
--

local setmetatable = setmetatable

local mdp = require"mdp"

local zmq = require"zmq"
local zpoller = require"zmq.poller"
local zmsg = require"zmsg"
require"zhelpers"

local s_version_assert = s_version_assert

local obj_mt = {}
obj_mt.__index = obj_mt

function obj_mt:set_timeout(timeout)
    self.timeout = timeout
end

function obj_mt:set_retries(retries)
    self.retries = retries
end

function obj_mt:destroy()
    if self.client then self.client:close() end
    self.context:term()
```

```
end

local function s_mdcli_connect_to_broker(self)
    -- close old socket.
    if self.client then
        self.poller:remove(self.client)
        self.client:close()
    end
    self.client = assert(self.context:socket(zmq.REQ))
    assert(self.client:setopt(zmq.LINGER, 0))
    assert(self.client:connect(self.broker))
    if self.verbose then
        s_console("I: connecting to broker at %s...", self.broker)
    end
    -- add socket to poller
    self.poller:add(self.client, zmq.POLLIN, function()
        self.got_reply = true
    end)
end

--
-- Send request to broker and get reply by hook or crook
-- Returns the reply message or nil if there was no reply.
--
function obj_mt:send(service, request)
    -- Prefix request with protocol frames
    -- Frame 1: "MDPCxy" (six bytes, MDP/Client x.y)
    -- Frame 2: Service name (printable string)
    request:push(service)
    request:push(mdp.MDPC_CLIENT)
    if self.verbose then
        s_console("I: send request to '%s' service:", service)
        request:dump()
    end

    local retries = self.retries
    while (retries > 0) do
        local msg = request:dup()
        msg:send(self.client)
        self.got_reply = false

        while true do
            local cnt = assert(self.poller:poll(self.timeout * 1000))
            if cnt ~= 0 and self.got_reply then
                local msg = zmsg.recv(self.client)
                if self.verbose then
                    s_console("I: received reply:")
                    msg:dump()
                end
                assert(msg:parts() >= 3)

                local header = msg:pop()
                assert(header == mdp.MDPC_CLIENT)
```

```
                local reply_service = msg:pop()
                assert(reply_service == service)
                return msg
            else
                retries = retries - 1
                if (retries > 0) then
                    if self.verbose then
                        s_console("W: no reply, reconnecting...")
                    end
                    -- Reconnect
                    s_mdcli_connect_to_broker(self)
                    break -- outer loop will resend request.
                else
                    if self.verbose then
                        s_console("W: permanent error, abandoning request")
                    end
                    return nil -- Giving up
                end
            end
        end
    end
end

module(...)

function new(broker, verbose)
    s_version_assert (2, 1);
    local self = setmetatable({
        context = zmq.init(1),
        poller = zpoller.new(1),
        broker = broker,
        verbose = verbose,
        timeout = 2500, -- msecs
        retries = 3,    -- before we abandon
    }, obj_mt)

    s_mdcli_connect_to_broker(self)
    return self
end

setmetatable(_M, { __call = function(self, ...) return new(...) end })
```

With an example test program that does 100K request-reply cycles:

Example 4-8. Majordomo client application (mdclient.lua)

```
--
--  Majordomo Protocol client example
--  Uses the mdcli API to hide all MDP aspects
--
--  Author: Robert G. Jakabosky <bobby@sharedrealm.com>
--
```

```
require"mdcliapi"
require"zmsg"
require"zhelpers"

local verbose = (arg[1] == "-v")
local session = mdcliapi.new("tcp://localhost:5555", verbose)

local count=1
repeat
    local request = zmsg.new("Hello world")
    local reply = session:send("echo", request)
    if not reply then
        break    --  Interrupt or failure
    end
    count = count + 1
until (count == 100000)
printf("%d requests/replies processed\n", count)
session:destroy()
```

And here is the worker API:

Example 4-9. Majordomo worker API (mdwrkapi.lua)

```
--
-- mdwrkapi.lua - Majordomo Protocol Worker API
--
-- Author: Robert G. Jakabosky <bobby@sharedrealm.com>
--

local HEARTBEAT_LIVENESS = 3  -- 3-5 is reasonable

local setmetatable = setmetatable

local mdp = require"mdp"

local zmq = require"zmq"
local zpoller = require"zmq.poller"
local zmsg = require"zmsg"
require"zhelpers"

local s_version_assert = s_version_assert

local obj_mt = {}
obj_mt.__index = obj_mt

function obj_mt:set_heartbeat(heartbeat)
    self.heartbeat = heartbeat
end

function obj_mt:set_reconnect(reconnect)
    self.reconnect = reconnect
```

```
end

function obj_mt:destroy()
    if self.worker then self.worker:close() end
    self.context:term()
end

-- Send message to broker
-- If no msg is provided, create one internally
local function s_mdwrk_send_to_broker(self, command, option, msg)
    msg = msg or zmsg.new()

    -- Stack protocol envelope to start of message
    if option then
        msg:push(option)
    end
    msg:push(command)
    msg:push(mdp.MDPW_WORKER)
    msg:push("")

    if self.verbose then
        s_console("I: sending %s to broker", mdp.mdps_commands[command])
        msg:dump()
    end
    msg:send(self.worker)
end

local function s_mdwrk_connect_to_broker(self)
    -- close old socket.
    if self.worker then
        self.poller:remove(self.worker)
        self.worker:close()
    end
    self.worker = assert(self.context:socket(zmq.DEALER))
    assert(self.worker:setopt(zmq.LINGER, 0))
    assert(self.worker:connect(self.broker))
    if self.verbose then
        s_console("I: connecting to broker at %s...", self.broker)
    end
    -- Register service with broker
    s_mdwrk_send_to_broker(self, mdp.MDPW_READY, self.service)
    -- If liveness hits zero, queue is considered disconnected
    self.liveness = HEARTBEAT_LIVENESS
    self.heartbeat_at = s_clock() + self.heartbeat
    -- add socket to poller
    self.poller:add(self.worker, zmq.POLLIN, function()
        self.got_msg = true
    end)
end

--
-- Send reply, if any, to broker and wait for next request.
--
```

```
function obj_mt:recv(reply)
    -- Format and send the reply if we are provided one
    if reply then
        assert(self.reply_to)
        reply:wrap(self.reply_to, "")
        self.reply_to = nil
        s_mdwrk_send_to_broker(self, mdp.MDPW_REPLY, nil, reply)
    end
    self.expect_reply = true

    self.got_msg = false
    while true do
        local cnt = assert(self.poller:poll(self.heartbeat * 1000))
        if cnt ~= 0 and self.got_msg then
            self.got_msg = false
            local msg = zmsg.recv(self.worker)
            if self.verbose then
                s_console("I: received message from broker:")
                msg:dump()
            end
            self.liveness = HEARTBEAT_LIVENESS
            -- Don't try to handle errors, just assert noisily
            assert(msg:parts() >= 3)

            local empty = msg:pop()
            assert(empty == "")

            local header = msg:pop()
            assert(header == mdp.MDPW_WORKER)

            local command = msg:pop()
            if command == mdp.MDPW_REQUEST then
                -- 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 = msg:unwrap()
                return msg -- We have a request to process
            elseif command == mdp.MDPW_HEARTBEAT then
                -- Do nothing for heartbeats
            elseif command == mdp.MDPW_DISCONNECT then
                -- dis-connect and re-connect to broker.
                s_mdwrk_connect_to_broker(self)
            else
                s_console("E: invalid input message (%d)", command:byte(1,1))
                msg:dump()
            end
        else
            self.liveness = self.liveness - 1
            if (self.liveness == 0) then
                if self.verbose then
                    s_console("W: disconnected from broker - retrying...")
                end
                -- sleep then Reconnect
                s_sleep(self.reconnect)
```

```
                s_mdwrk_connect_to_broker(self)
            end

            -- Send HEARTBEAT if it's time
            if (s_clock() > self.heartbeat_at) then
                s_mdwrk_send_to_broker(self, mdp.MDPW_HEARTBEAT)
                self.heartbeat_at = s_clock() + self.heartbeat
            end
        end
    end
end

module(...)

function new(broker, service, verbose)
    s_version_assert(2, 1);
    local self = setmetatable({
        context = zmq.init(1),
        poller = zpoller.new(1),
        broker = broker,
        service = service,
        verbose = verbose,
        heartbeat = 2500, -- msecs
        reconnect = 2500, -- msecs
    }, obj_mt)

    s_mdwrk_connect_to_broker(self)
    return self
end

setmetatable(_M, { __call = function(self, ...) return new(...) end })
```

With an example test program that implements an 'echo' service:

Example 4-10. Majordomo worker application (mdworker.lua)

```
--
--  Majordomo Protocol worker example
--  Uses the mdwrk API to hide all MDP aspects
--
--  Author: Robert G. Jakabosky <bobby@sharedrealm.com>
--

require"mdwrkapi"
require"zmsg"

local verbose = (arg[1] == "-v")
local session = mdwrkapi.new("tcp://localhost:5555", "echo", verbose)

local reply
while true do
    local request = session:recv(reply)
```

```
    if not request then
        break              --  Worker was interrupted
    end
    reply = request        --  Echo is complex... :-)
end
session:destroy()
```

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 back-off, 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.

You might wonder why the worker API is manually closing its socket and opening a new one, when ØMQ will automatically reconnect a socket if the peer disappears and comes back. Look back at the Simple Pirate worker, and the Paranoid Pirate worker to understand. While ØMQ will automatically reconnect workers, if the broker dies and comes back up, this isn't sufficient to re-register the workers with the broker. There are at least two solutions I know of. The simplest, which we use here, is that the worker monitors the connection using heartbeats, and if it decides the broker is dead, closes its socket and starts afresh with a new socket. The alternative is for the broker to challenge unknown workers -- when it gets a heartbeat from the worker -- and ask them to re-register. That would require protocol support.

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.

And here is the broker:

Example 4-11. Majordomo broker (mdbroker.lua)

```
--
--  Majordomo Protocol broker
--  A minimal implementation of http://rfc.zeromq.org/spec:7 and spec:8
--
--  Author: Robert G. Jakabosky <bobby@sharedrealm.com>
--
require"zmq"
require"zmq.poller"
require"zmsg"
require"zhelpers"
require"mdp"

local tremove = table.remove
```

```
--  We'd normally pull these from config data

local HEARTBEAT_LIVENESS   = 3       --  3-5 is reasonable
local HEARTBEAT_INTERVAL   = 2500    --  msecs
local HEARTBEAT_EXPIRY     = HEARTBEAT_INTERVAL * HEARTBEAT_LIVENESS

--  ---------------------------------------------------------------------
--  Constructor for broker object

--  ---------------------------------------------------------------------
--  Broker object's metatable.
local broker_mt = {}
broker_mt.__index = broker_mt

function broker_new(verbose)
    local context = zmq.init(1)
    --  Initialize broker state
    return setmetatable({
        context = context,
        socket = context:socket(zmq.ROUTER),
        verbose = verbose,
        services = {},
        workers = {},
        waiting = {},
        heartbeat_at = s_clock() + HEARTBEAT_INTERVAL,
    }, broker_mt)
end

--  ---------------------------------------------------------------------
--  Service object
local service_mt = {}
service_mt.__index = service_mt

--  Worker object
local worker_mt = {}
worker_mt.__index = worker_mt

-- helper list remove function
local function zlist_remove(list, item)
    for n=#list,1,-1 do
        if list[n] == item then
            tremove(list, n)
        end
    end
end

--  ---------------------------------------------------------------------
--  Destructor for broker object

function broker_mt:destroy()
    self.socket:close()
    self.context:term()
```

```
    for name, service in pairs(self.services) do
        service:destroy()
    end
    for id, worker in pairs(self.workers) do
        worker:destroy()
    end
end

--  ---------------------------------------------------------------------
--  Bind broker to endpoint, can call this multiple times
--  We use a single socket for both clients and workers.

function broker_mt:bind(endpoint)
    self.socket:bind(endpoint)
    s_console("I: MDP broker/0.1.1 is active at %s", endpoint)
end

--  ---------------------------------------------------------------------
--  Delete any idle workers that haven't pinged us in a while.

function broker_mt:purge_workers()
    local waiting = self.waiting
    for n=1,#waiting do
        local worker = waiting[n]
        if (worker:expired()) then
            if (self.verbose) then
                s_console("I: deleting expired worker: %s", worker.identity)
            end

            self:worker_delete(worker, false)
        end
    end
end

--  ---------------------------------------------------------------------
--  Locate or create new service entry

function broker_mt:service_require(name)
    assert (name)
    local service = self.services[name]
    if not service then
        service = setmetatable({
            name = name,
            requests = {},
            waiting = {},
            workers = 0,
        }, service_mt)
        self.services[name] = service
        if (self.verbose) then
            s_console("I: received message:")
        end
    end
    return service
```

```
end

--  ---------------------------------------------------------------------
--  Destroy service object, called when service is removed from
--  broker.services.

function service_mt:destroy()
end

--  ---------------------------------------------------------------------
--  Dispatch requests to waiting workers as possible

function broker_mt:service_dispatch(service, msg)
    assert (service)
    local requests = service.requests
    if (msg) then               --  Queue message if any
        requests[#requests + 1] = msg
    end

    self:purge_workers()
    local waiting = service.waiting
    while (#waiting > 0 and #requests > 0) do
        local worker = tremove(waiting, 1) -- pop worker from service's waiting queue.
        zlist_remove(self.waiting, worker) -- also remove worker from broker's waiting queue
        local msg = tremove(requests, 1) -- pop request from service's request queue.
        self:worker_send(worker, mdp.MDPW_REQUEST, nil, msg)
    end
end

--  ---------------------------------------------------------------------
--  Handle internal service according to 8/MMI specification

function broker_mt:service_internal(service_name, msg)
    if (service_name == "mmi.service") then
        local name = msg:body()
        local service = self.services[name]
        if (service and service.workers) then
            msg:body_set("200")
        else
            msg:body_set("404")
        end
    else
        msg:body_set("501")
    end

    --  Remove & save client return envelope and insert the
    --  protocol header and service name, then rewrap envelope.
    local client = msg:unwrap()
    msg:wrap(mdp.MDPC_CLIENT, service_name)
    msg:wrap(client, "")

    msg:send(self.socket)
end
```

```
--  ---------------------------------------------------------------------
--  Creates worker if necessary

function broker_mt:worker_require(identity)
    assert (identity)

    --  self.workers is keyed off worker identity
    local worker = self.workers[identity]
    if (not worker) then
        worker = setmetatable({
            identity = identity,
            expiry = 0,
        }, worker_mt)
        self.workers[identity] = worker
        if (self.verbose) then
            s_console("I: registering new worker: %s", identity)
        end
    end
    return worker
end

--  ---------------------------------------------------------------------
--  Deletes worker from all data structures, and destroys worker

function broker_mt:worker_delete(worker, disconnect)
    assert (worker)
    if (disconnect) then
        self:worker_send(worker, mdp.MDPW_DISCONNECT)
    end
    local service = worker.service
    if (service) then
        zlist_remove (service.waiting, worker)
        service.workers = service.workers - 1
    end
    zlist_remove (self.waiting, worker)
    self.workers[worker.identity] = nil
    worker:destroy()
end

--  ---------------------------------------------------------------------
--  Destroy worker object, called when worker is removed from
--  broker.workers.

function worker_mt:destroy(argument)
end

--  ---------------------------------------------------------------------
--  Process message sent to us by a worker

function broker_mt:worker_process(sender, msg)
    assert (msg:parts() >= 1)     --  At least, command
```

```
    local command = msg:pop()
    local worker_ready = (self.workers[sender] ~= nil)
    local worker = self:worker_require(sender)

    if (command == mdp.MDPW_READY) then
        if (worker_ready) then          --  Not first command in session then
            self:worker_delete(worker, true)
        elseif (sender:sub(1,4) == "mmi.") then  --  Reserved service name
            self:worker_delete(worker, true)
        else
            --  Attach worker to service and mark as idle
            local service_name = msg:pop()
            local service = self:service_require(service_name)
            worker.service = service
            service.workers = service.workers + 1
            self:worker_waiting(worker)
        end
    elseif (command == mdp.MDPW_REPLY) then
        if (worker_ready) then
            --  Remove & save client return envelope and insert the
            --  protocol header and service name, then rewrap envelope.
            local client = msg:unwrap()
            msg:wrap(mdp.MDPC_CLIENT, worker.service.name)
            msg:wrap(client, "")

            msg:send(self.socket)
            self:worker_waiting(worker)
        else
            self:worker_delete(worker, true)
        end
    elseif (command == mdp.MDPW_HEARTBEAT) then
        if (worker_ready) then
            worker.expiry = s_clock() + HEARTBEAT_EXPIRY
        else
            self:worker_delete(worker, true)
        end
    elseif (command == mdp.MDPW_DISCONNECT) then
        self:worker_delete(worker, false)
    else
        s_console("E: invalid input message (%d)", command:byte(1,1))
        msg:dump()
    end
end

--  ---------------------------------------------------------------------
--  Send message to worker
--  If pointer to message is provided, sends & destroys that message

function broker_mt:worker_send(worker, command, option, msg)
    msg = msg and msg:dup() or zmsg.new()

    --  Stack protocol envelope to start of message
    if (option) then                --  Optional frame after command
```

```
        msg:push(option)
    end
    msg:push(command)
    msg:push(mdp.MDPW_WORKER)
    --  Stack routing envelope to start of message
    msg:wrap(worker.identity, "")

    if (self.verbose) then
        s_console("I: sending %s to worker", mdp.mdps_commands[command])
        msg:dump()
    end
    msg:send(self.socket)
end

--  ---------------------------------------------------------------------
--  This worker is now waiting for work

function broker_mt:worker_waiting(worker)
    --  Queue to broker and service waiting lists
    self.waiting[#self.waiting + 1] = worker
    worker.service.waiting[#worker.service.waiting + 1] = worker
    worker.expiry = s_clock() + HEARTBEAT_EXPIRY
    self:service_dispatch(worker.service, nil)
end

--  ---------------------------------------------------------------------
--  Return 1 if worker has expired and must be deleted

function worker_mt:expired()
    return (self.expiry < s_clock())
end
--  ---------------------------------------------------------------------
--  Process a request coming from a client

function broker_mt:client_process(sender, msg)
    assert (msg:parts() >= 2)     --  Service name + body

    local service_name = msg:pop()
    local service = self:service_require(service_name)
    --  Set reply return address to client sender
    msg:wrap(sender, "")
    if (service_name:sub(1,4) == "mmi.") then
        self:service_internal(service_name, msg)
    else
        self:service_dispatch(service, msg)
    end
end


--  ---------------------------------------------------------------------
--  Main broker work happens here

local verbose = (arg[1] == "-v")
```

```
s_version_assert (2, 1)
s_catch_signals ()
local self = broker_new(verbose)
self:bind("tcp://*:5555")

local poller = zmq.poller.new(1)

--  Process next input message, if any
poller:add(self.socket, zmq.POLLIN, function()
    local msg = zmsg.recv(self.socket)
    if (self.verbose) then
        s_console("I: received message:")
        msg:dump()
    end
    local sender = msg:pop()
    local empty  = msg:pop()
    local header = msg:pop()

    if (header == mdp.MDPC_CLIENT) then
        self:client_process(sender, msg)
    elseif (header == mdp.MDPW_WORKER) then
        self:worker_process(sender, msg)
    else
        s_console("E: invalid message:")
        msg:dump()
    end
end)

--  Get and process messages forever or until interrupted
while (not s_interrupted) do
    local cnt = assert(poller:poll(HEARTBEAT_INTERVAL * 1000))
    --  Disconnect and delete any expired workers
    --  Send heartbeats to idle workers if needed
    if (s_clock() > self.heartbeat_at) then
        self:purge_workers()
        local waiting = self.waiting
        for n=1,#waiting do
            local worker = waiting[n]
            self:worker_send(worker, mdp.MDPW_HEARTBEAT)
        end
        self.heartbeat_at = s_clock() + HEARTBEAT_INTERVAL
    end
end
if (s_interrupted) then
    printf("W: interrupt received, shutting down...\n")
end
self:destroy()
```

This is by far the most complex example we've seen. It's almost 500 lines of code. To write this, and make it somewhat 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-fail-over 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.
- We later improved and extended the protocol and the Majordomo implementation, which now sits in its own Github project. If you want a properly usable Majordomo stack, use the github project.

4.9. Asynchronous Majordomo Pattern

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 doing network round-trips. ØMQ disables Nagle's algorithm (http://en.wikipedia.org/wiki/Nagles_algorithm), but round-tripping is still slow.

Theory is great in theory, but in practice, practice is better. Let's measure the actual 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 mock-up a client, broker, and worker:

Example 4-12. Round-trip demonstrator (tripping.lua)

```
--
--  Round-trip demonstrator
--
--  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.
--
--  Author: Robert G. Jakabosky <bobby@sharedrealm.com>
--
require"zmq"
```

```
require"zmq.threads"
require"zmsg"

local common_code = [[
    require"zmq"
    require"zmsg"
    require"zhelpers"
]]

local client_task = common_code .. [[
    local context = zmq.init(1)
    local client = context:socket(zmq.DEALER)
    client:setopt(zmq.IDENTITY, "C", 1)
    client:connect("tcp://localhost:5555")

    printf("Setting up test...\n")
    s_sleep(100)

    local requests
    local start

    printf("Synchronous round-trip test...\n")
    requests = 10000
    start = s_clock()
    for n=1,requests do
        local msg = zmsg.new("HELLO")
        msg:send(client)
        msg = zmsg.recv(client)
    end
    printf(" %d calls/second\n",
        (1000 * requests) / (s_clock() - start))

    printf("Asynchronous round-trip test...\n")
    requests = 100000
    start = s_clock()
    for n=1,requests do
        local msg = zmsg.new("HELLO")
        msg:send(client)
    end
    for n=1,requests do
        local msg = zmsg.recv(client)
    end
    printf(" %d calls/second\n",
        (1000 * requests) / (s_clock() - start))

    client:close()
    context:term()
]]

local worker_task = common_code .. [[
    local context = zmq.init(1)
    local worker = context:socket(zmq.DEALER)
    worker:setopt(zmq.IDENTITY, "W", 1)
```

```
    worker:connect("tcp://localhost:5556")

    while true do
        local msg = zmsg.recv(worker)
        msg:send(worker)
    end
    worker:close()
    context:term()
]]

local broker_task = common_code .. [[
    --  Prepare our context and sockets
    local context = zmq.init(1)
    local frontend = context:socket(zmq.ROUTER)
    local backend  = context:socket(zmq.ROUTER)
    frontend:bind("tcp://*:5555")
    backend:bind("tcp://*:5556")

    require"zmq.poller"
    local poller = zmq.poller(2)
    poller:add(frontend, zmq.POLLIN, function()
        local msg = zmsg.recv(frontend)
        --msg[1] = "W"
        msg:pop()
        msg:push("W")
        msg:send(backend)
    end)
    poller:add(backend, zmq.POLLIN, function()
        local msg = zmsg.recv(backend)
        --msg[1] = "C"
        msg:pop()
        msg:push("C")
        msg:send(frontend)
    end)
    poller:start()
    frontend:close()
    backend:close()
    context:term()
]]

s_version_assert(2, 1)

local client = zmq.threads.runstring(nil, client_task)
assert(client:start())
local worker = zmq.threads.runstring(nil, worker_task)
assert(worker:start(true))
local broker = zmq.threads.runstring(nil, broker_task)
assert(broker:start(true))

assert(client:join())
```

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 to make it faster.

First, we 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:

Example 4-13. Majordomo asynchronous client API (mdcliapi2.lua)

```
--
-- mdcliapi2.lua - Majordomo Protocol Client API (async version)
--
-- Author: Robert G. Jakabosky <bobby@sharedrealm.com>
--

local setmetatable = setmetatable

local mdp = require"mdp"

local zmq = require"zmq"
local zpoller = require"zmq.poller"
local zmsg = require"zmsg"
require"zhelpers"

local s_version_assert = s_version_assert

local obj_mt = {}
obj_mt.__index = obj_mt

function obj_mt:set_timeout(timeout)
    self.timeout = timeout
end
```

```
function obj_mt:destroy()
    if self.client then self.client:close() end
    self.context:term()
end

local function s_mdcli_connect_to_broker(self)
    -- close old socket.
    if self.client then
        self.poller:remove(self.client)
        self.client:close()
    end
    self.client = assert(self.context:socket(zmq.DEALER))
    assert(self.client:setopt(zmq.LINGER, 0))
    assert(self.client:connect(self.broker))
    if self.verbose then
        s_console("I: connecting to broker at %s...", self.broker)
    end
    -- add socket to poller
    self.poller:add(self.client, zmq.POLLIN, function()
        self.got_reply = true
    end)
end

--
-- Send request to broker and get reply by hook or crook
--
function obj_mt:send(service, request)
    -- 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)
    request:push(service)
    request:push(mdp.MDPC_CLIENT)
    request:push("")
    if self.verbose then
        s_console("I: send request to '%s' service:", service)
        request:dump()
    end
    request:send(self.client)
    return 0
end

--  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...
function obj_mt:recv()
    self.got_reply = false

    local cnt = assert(self.poller:poll(self.timeout * 1000))
    if cnt ~= 0 and self.got_reply then
        local msg = zmsg.recv(self.client)
        if self.verbose then
            s_console("I: received reply:")
```

```
            msg:dump()
        end
        assert(msg:parts() >= 3)

        local empty = msg:pop()
        assert(empty == "")

        local header = msg:pop()
        assert(header == mdp.MDPC_CLIENT)

        return msg
    end
    if self.verbose then
        s_console("W: permanent error, abandoning request")
    end
    return nil -- Giving up
end

module(...)

function new(broker, verbose)
    s_version_assert (2, 1);
    local self = setmetatable({
        context = zmq.init(1),
        poller = zpoller.new(1),
        broker = broker,
        verbose = verbose,
        timeout = 2500, -- msecs
    }, obj_mt)

    s_mdcli_connect_to_broker(self)
    return self
end

setmetatable(_M, { __call = function(self, ...) return new(...) end })
```

And here's the corresponding client test program:

Example 4-14. Majordomo client application (mdclient2.lua)

```
--
--  Majordomo Protocol client example - asynchronous
--  Uses the mdcli API to hide all MDP aspects
--
--  Author: Robert G. Jakabosky <bobby@sharedrealm.com>
--

require"mdcliapi2"
require"zmsg"
require"zhelpers"

local verbose = (arg[1] == "-v")
```

```
local session = mdcliapi2.new("tcp://localhost:5555", verbose)

local count=100000
for n=1,count do
    local request = zmsg.new("Hello world")
    session:send("echo", request)
end
for n=1,count do
    local reply = session:recv()
    if not reply then
        break   --  Interrupted by Ctrl-C
    end
end
printf("%d replies received\n", count)
session:destroy()
```

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 PC, 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 more 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 fail-over, 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.

4.10. Service Discovery

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.

Another 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 returned requests from 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:

Example 4-15. Service discovery over Majordomo (mmiecho.lua)

```
--
--  MMI echo query example
--
```

```
--  Author: Robert G. Jakabosky <bobby@sharedrealm.com>
--

require"mdcliapi"
require"zmsg"
require"zhelpers"

local verbose = (arg[1] == "-v")
local session = mdcliapi.new("tcp://localhost:5555", verbose)

--  This is the service we want to look up
local request = zmsg.new("echo")

--  This is the service we send our request to
local reply = session:send("mmi.service", request)

if (reply) then
    printf ("Lookup echo service: %s\n", reply:body())
else
    printf ("E: no response from broker, make sure it's running\n")
end

session:destroy()
```

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.

4.11. Idempotent Services

Idempotency is not something you 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 children 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 it's 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.

4.12. Disconnected Reliability (Titanic Pattern)

Once you realize that Majordomo is a 'reliable' message broker, you might be tempted to add some spinning rust (that is, ferrous-based hard disk platters). 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 brutal cynicism is 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 disconnected 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.

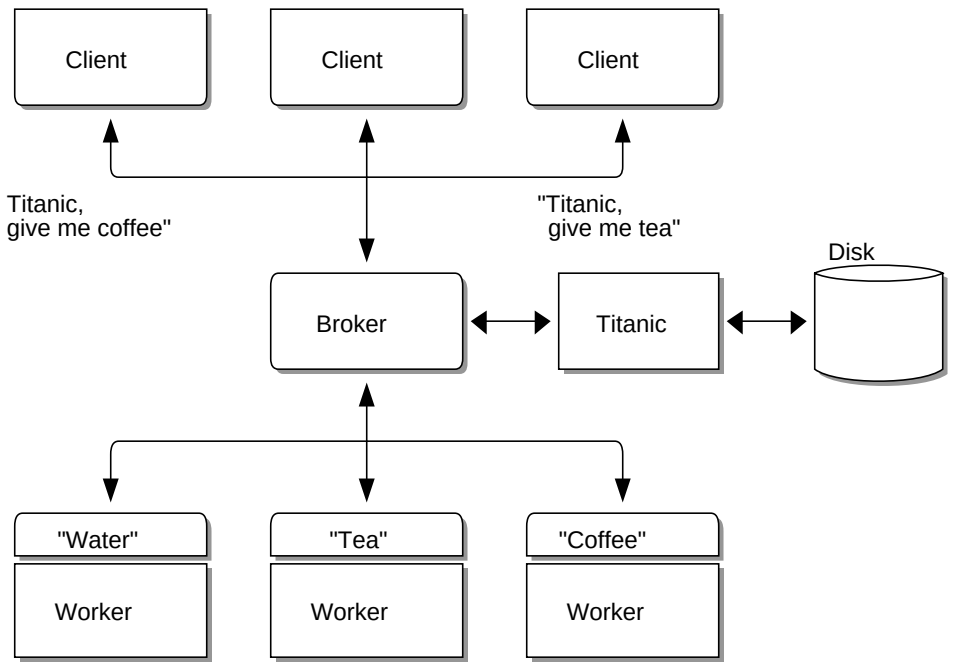
So, here's the Titanic pattern(Figure 4-5), in which we write messages to disk to ensure they never get lost, no matter how sporadically clients and workers are connected. As we did for service discovery, we're going to layer Titanic on top of Majordomo rather than extend MDP. It's wonderfully lazy because 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 is *much* easier because we divide and conquer: the broker handles message routing and the worker handles reliability.
- 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 4-5. The 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?

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:

Example 4-16. Titanic client example (ticlient.lua)

```
(This example still needs translation into Lua)
```

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:

Example 4-17. Titanic broker example (titanic.lua)

```
(This example still needs translation into Lua)
```

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.

4.13. High-availability Pair (Binary Star Pattern)

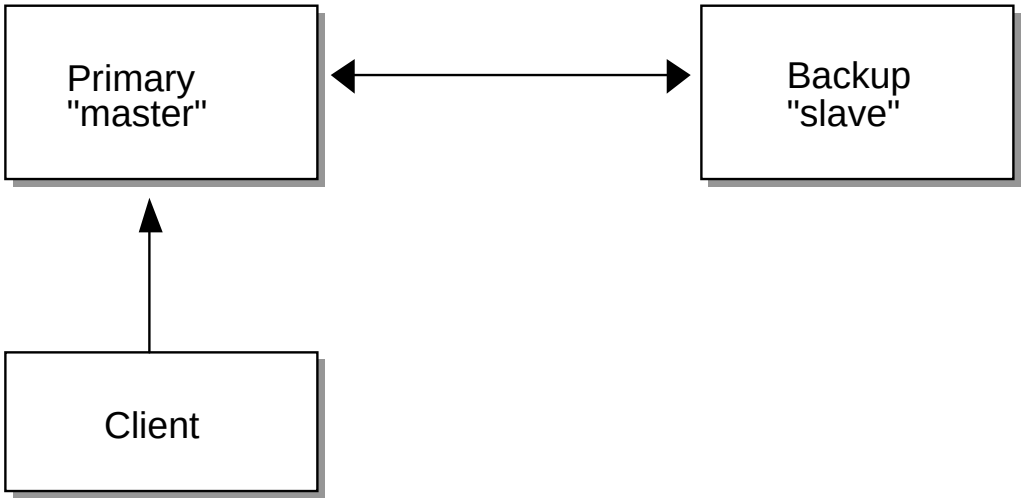
4.13.1. Overview

The Binary Star pattern puts two servers in a primary-backup high-availability pair(Figure 4-6). 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 (http://www.openamq.org). We designed it:

- To provide a straight-forward high-availability solution.
- To be simple enough to actually understand and use.
- To fail-over reliably when needed, and only when needed.

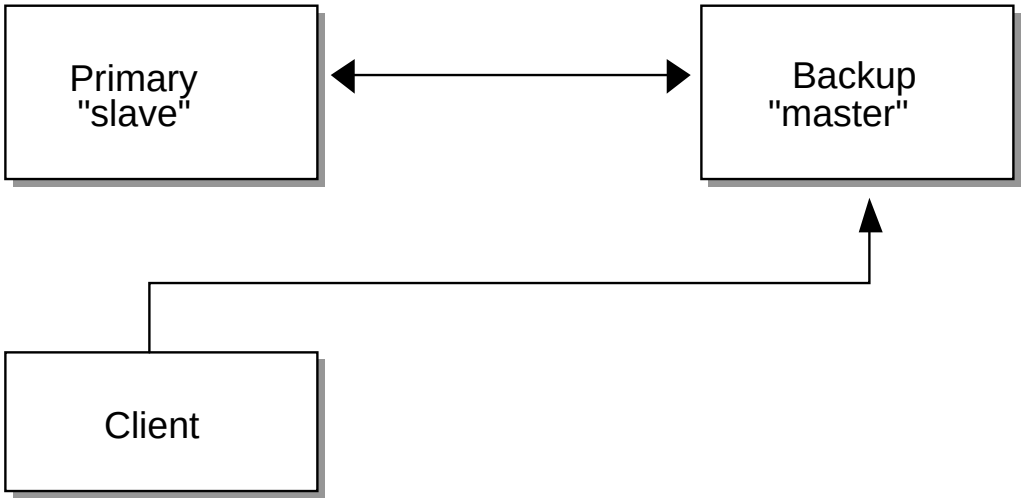
Figure 4-6. High-availability Pair, Normal Operation



Assuming we have a Binary Star pair running, here are the different scenarios that will result in fail-over happening(Figure 4-7):

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.

Figure 4-7. High-availability Pair During Failover



Recovery from fail-over 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.

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 fail-over timeout will cause applications to disconnect, then reconnect, then disconnect again, which may disturb users.

4.13.2. Detailed Requirements

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 fail-over 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, fail-over synchronization must be able to use 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 servers 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 at most 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 fail-over 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 fail-over 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 fail-over. In our example, the fail-over 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 fail-over timeout.
5. Recreate all of the state they require on a server.
6. Retransmit messages lost during a fail-over, 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 fail-over.

4.13.3. Preventing Split-Brain Syndrome

"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 fail-over 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 fail-over, 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.

4.13.4. Binary Star Implementation

Without further ado, here is a proof-of-concept implementation of the Binary Star server:

Example 4-18. Binary Star server (bstarsrv.lua)

```
(This example still needs translation into Lua)
```

And here is the client:

Example 4-19. Binary Star client (bstarcli.lua)

```
(This example still needs translation into Lua)
```

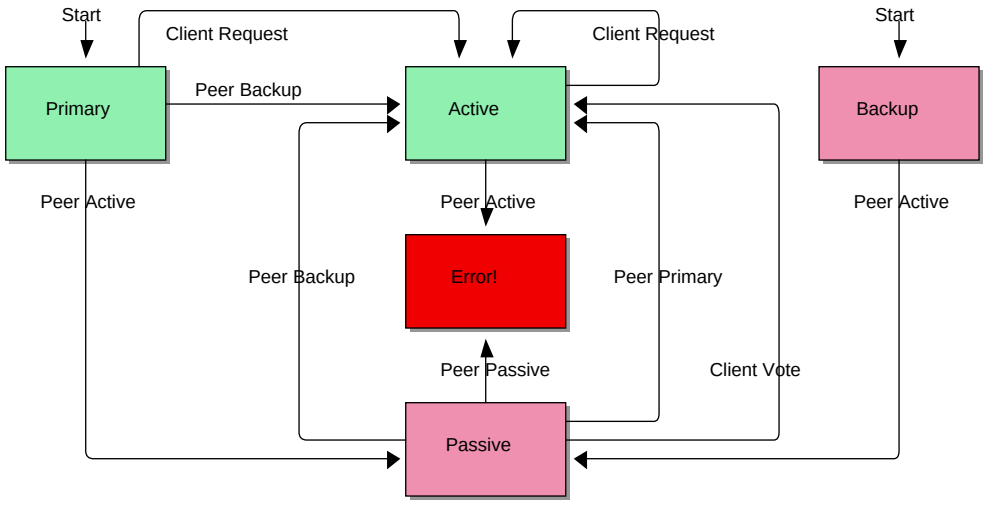
To test Binary Star, start the servers and client in any order:

```
bstarsrv -p     # Start primary
bstarsrv -b     # Start backup
bstarcli
```

You can then provoke fail-over by killing the primary server, and recovery by restarting the primary and killing the backup. Note how it’s the client vote that triggers fail-over, and recovery.

Binary star is driven by a finite state machine(Figure 4-8). 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 received a client request. "Client Vote" means we’ve received a client request AND our peer is inactive for two heartbeats.

Figure 4-8. 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 fail-over.

4.13.5. Binary Star Reactor

Binary Star is useful and generic enough to package up as a reusable reactor class. The reactor then runs and calls our code whenever it has a message to process. This is much nicer than copying/pasting the Binary Star code into each server where we want that capability. In C we wrap the CZMQ `zloop` class, though your mileage 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:

Example 4-20. Binary Star core class (bstar.lua)

```
(This example still needs translation into Lua)
```

Which gives us the following short main program for the server:

Example 4-21. Binary Star server, using core class (bstarsrv2.lua)

```
(This example still needs translation into Lua)
```

4.14. Brokerless Reliability (Freelance Pattern)

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".

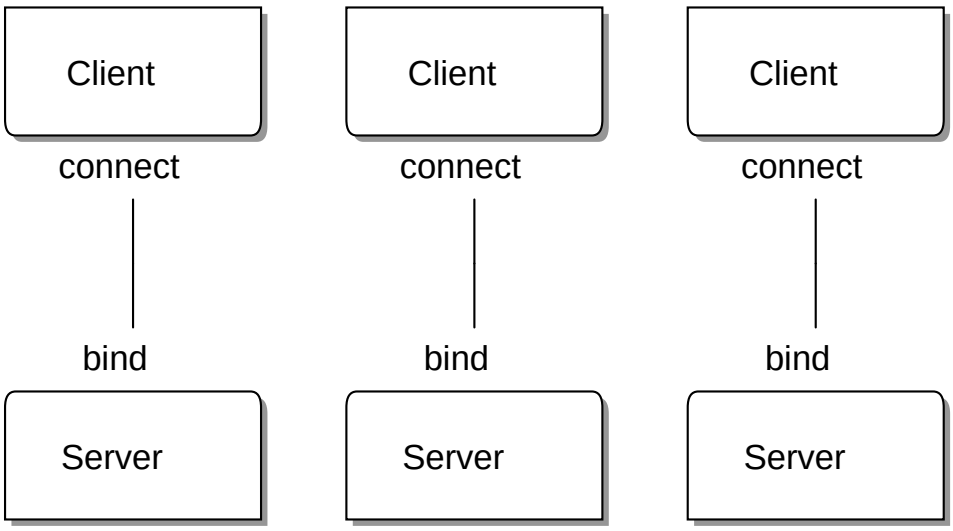
A ØMQ name service (and we'll make a simple implementation) has to:

- 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 configuration 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 4-9).

Figure 4-9. 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).

4.14.1. Model One - Simple Retry and Failover

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:

Example 4-22. Freelance server, Model One (flserver1.lua)

```
--
--  Freelance server - Model 1
--  Trivial echo service
```

```
--
--  Author: Robert G. Jakabosky <bobby@sharedrealm.com>
--
require"zmsg"
require"zmq"

if (#arg < 1) then
    printf("I: syntax: %s <endpoint>\n", arg[0])
    os.exit(0)
end
local context = zmq.init(1)
s_catch_signals()

--  Implement basic echo service
local server = context:socket(zmq.REP)
server:bind(arg[1])
printf("I: echo service is ready at %s\n", arg[1])
while (not s_interrupted) do
    local msg, err = zmsg.recv(server)
    if err then
        print('recv error:', err)
        break          --  Interrupted
    end
    msg:send(server)
end
if (s_interrupted) then
    printf("W: interrupted\n")
end
server:close()
context:term()
```

Then start the client, specifying one or more connect endpoints as arguments:

Example 4-23. Freelance client, Model One (flclient1.lua)

```
(This example still needs translation into Lua)
```

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 fail-over 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.

4.14.2. Model Two - Brutal Shotgun Massacre

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:

Example 4-24. Freelance server, Model Two (flserver2.lua)

```
--
--  Freelance server - Model 2
--  Does some work, replies OK, with message sequencing
--
--  Author: Robert G. Jakabosky <bobby@sharedrealm.com>
--
require"zmq"
require"zmsg"
```

```
if (#arg < 1) then
    printf ("I: syntax: %s <endpoint>\n", arg[0])
    os.exit (0)
end
local context = zmq.init(1)
s_catch_signals()

local server = context:socket(zmq.REP)
server:bind(arg[1])
printf ("I: service is ready at %s\n", arg[1])
while (not s_interrupted) do
    local msg, err = zmsg.recv(server)
    if err then
        print('recv error:', err)
        break          --  Interrupted
    end
    --  Fail nastily if run against wrong client
    assert (msg:parts() == 2)

    msg:body_set("OK")
    msg:send(server)
end
if (s_interrupted) then
    printf("W: interrupted\n")
end
server:close()
context:term()
```

Then start the client, specifying the connect endpoints as arguments:

Example 4-25. Freelance client, Model Two (flclient2.lua)

```
(This example still needs translation into Lua)
```

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 frame 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 fail-over, 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.

4.14.3. Model Three - Complex and Nasty

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 anonymous sockets (which haven't set an identity) is not possible. Both sides generate an identity (for the other peer) 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. Insane, complex, and nasty are great attributes for a genocidal dictator, but 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:

Example 4-26. Freelance server, Model Three (flserver3.lua)

```
--
--  Freelance server - Model 3
--  Uses an ROUTER/ROUTER socket but just one thread
--
--  Author: Robert G. Jakabosky <bobby@sharedrealm.com>
--
require"zmq"
require"zmsg"

local verbose = (arg[1] == "-v")

local context = zmq.init(1)
s_catch_signals ()

--  Prepare server socket with predictable identity
local bind_endpoint = "tcp://*:5555"
local connect_endpoint = "tcp://localhost:5555"
local server = context:socket(zmq.ROUTER)
server:setopt(zmq.IDENTITY, connect_endpoint)
server:bind(bind_endpoint)
printf ("I: service is ready at %s\n", bind_endpoint)

while (not s_interrupted) do
    local request = zmsg.recv (server)
    local reply = nil
    if (not request) then
        break          --  Interrupted
    end
    if (verbose) then
        request:dump()
```

```
    end
    --  Frame 0: identity of client
    --  Frame 1: PING, or client control frame
    --  Frame 2: request body
    local address = request:pop()
    if (request:parts() == 1 and request:body() == "PING") then
        reply = zmsg.new ("PONG")
    elseif (request:parts() > 1) then
        reply = request
        request = nil
        reply:body_set("OK")
    end
    reply:push(address)
    if (verbose and reply) then
        reply:dump()
    end
    reply:send(server)
end
if (s_interrupted) then
    printf ("W: interrupted\n")
end
server:close()
context:term()
```

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:

Example 4-27. Freelance client, Model Three (flclient3.lua)

```
(This example still needs translation into Lua)
```

And here, almost as complex and large as the Majordomo broker, is the client API class:

Example 4-28. Freelance client API (flcliapi.lua)

```
(This example still needs translation into Lua)
```

This API implementation is fairly sophisticated and uses a couple of techniques that we've not seen before:

Multithreaded API

The client API consists of two parts, a synchronous 'flcliapi' class that runs in the application thread, and an asynchronous 'agent' class that runs as a background thread. Remember how ØMQ makes it easy to create multithreaded apps. The flcliapi and agent classes talk to each other with messages over an `inproc` socket. All ØMQ aspects (such as creating and destroying 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.

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.

4.15. Conclusion

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 5. Advanced Publish-Subscribe

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 monitoring.

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).
- How to use reactors to simplify complex servers.
- How to use the Binary Star pattern to add failover to a server.
- How to monitor a publish-subscribe network (the *Espresso* pattern).

5.1. Slow Subscriber Detection (Suicidal Snail Pattern)

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:

Example 5-1. Suicidal Snail (suisnail.lua)

```
--
--  Suicidal Snail
--
--  Author: Robert G. Jakabosky <bobby@sharedrealm.com>
--
require"zmq"
require"zmq.threads"
require"zhelpers"

--  ---------------------------------------------------------------------
--  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.

local subscriber = [[
    require"zmq"
    require"zhelpers"

    local MAX_ALLOWED_DELAY    = 1000    --  msecs

    local context = zmq.init(1)

    --  Subscribe to everything
    local subscriber = context:socket(zmq.SUB)
    subscriber:connect("tcp://localhost:5556")
    subscriber:setopt(zmq.SUBSCRIBE, "", 0)

    --  Get and process messages
    while true do
        local msg = subscriber:recv()
        local clock = tonumber(msg)

        --  Suicide snail logic
        if (s_clock () - clock > MAX_ALLOWED_DELAY) then
            fprintf (io.stderr, "E: subscriber cannot keep up, aborting\n")
            break
        end
        --  Work for 1 msec plus some random additional time
        s_sleep (1 + randof (2))
    end
    subscriber:close()
    context:term()
]]

--  ---------------------------------------------------------------------
--  This is our server task
--  It publishes a time-stamped message to its pub socket every 1ms.

local publisher = [[
    require"zmq"
    require"zhelpers"

    local context = zmq.init(1)

    --  Prepare publisher
    local publisher = context:socket(zmq.PUB)
    publisher:bind("tcp://*:5556")

    while true do
        --  Send current clock (msecs) to subscribers
        publisher:send(tostring(s_clock()))
        s_sleep (1);            --  1msec wait
    end
    publisher:close()
```

```
    context:term()
]]

--  This main thread simply starts a client, and a server, and then
--  waits for the client to croak.
--

local server_thread = zmq.threads.runstring(nil, publisher)
server_thread:start(true)

local client_thread = zmq.threads.runstring(nil, subscriber)
client_thread:start()
client_thread:join()
```

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.

5.2. High-speed Subscribers (Black Box Pattern)

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.)

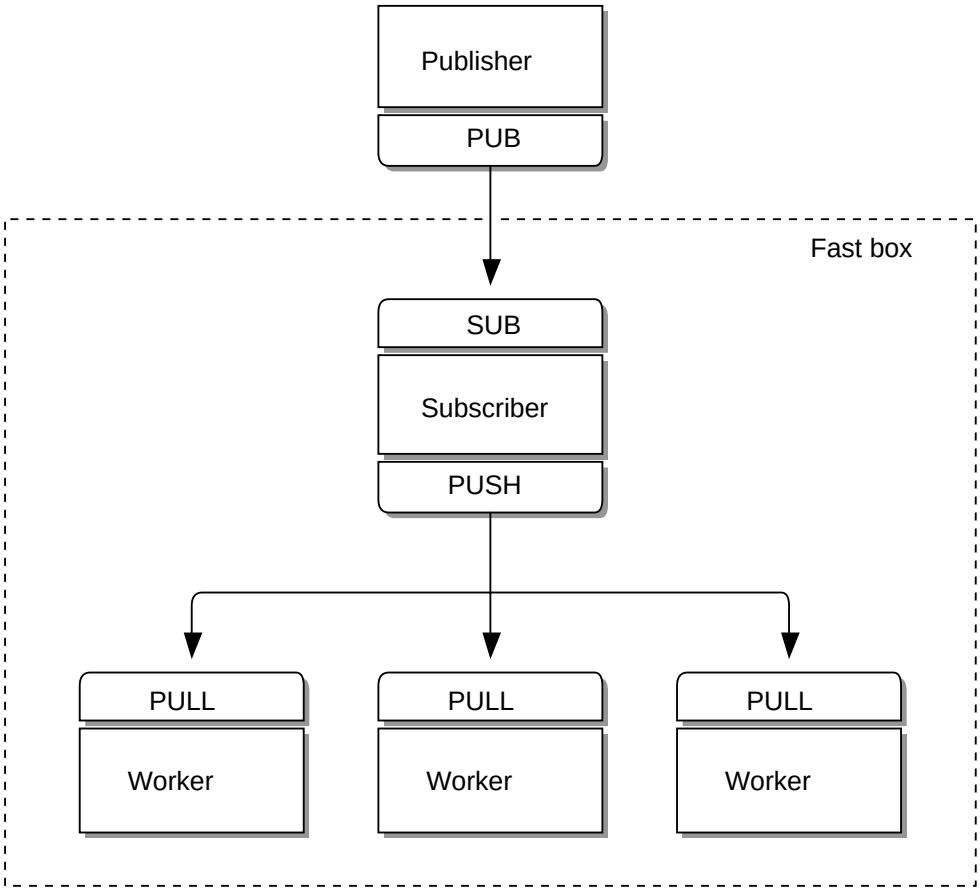
And as we pump data into our subscribers, we notice two things:

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(Figure 5-1). This is the simplest and fastest approach.

Figure 5-1. The Simple Black Box Pattern



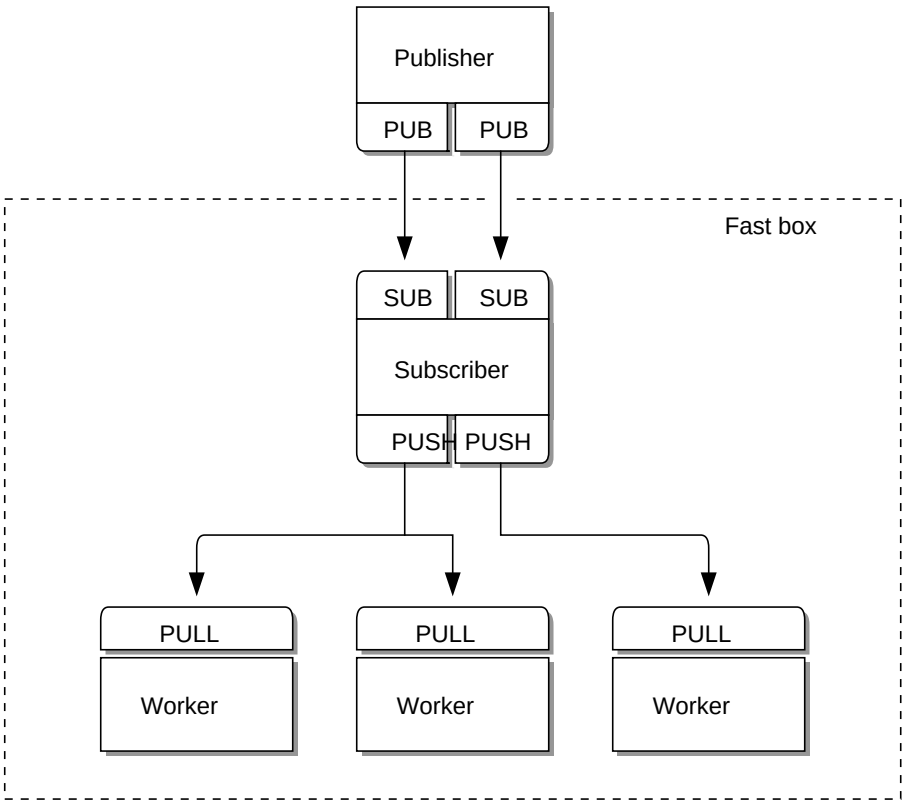
The subscriber talks to the publisher over TCP or PGM. The subscriber talks to its workers, which are all in the same process, over inproc.

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(Figure 5-2). 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 5-2. 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.

5.3. A Shared Key-Value Cache (Clone Pattern)

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 key-value 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:

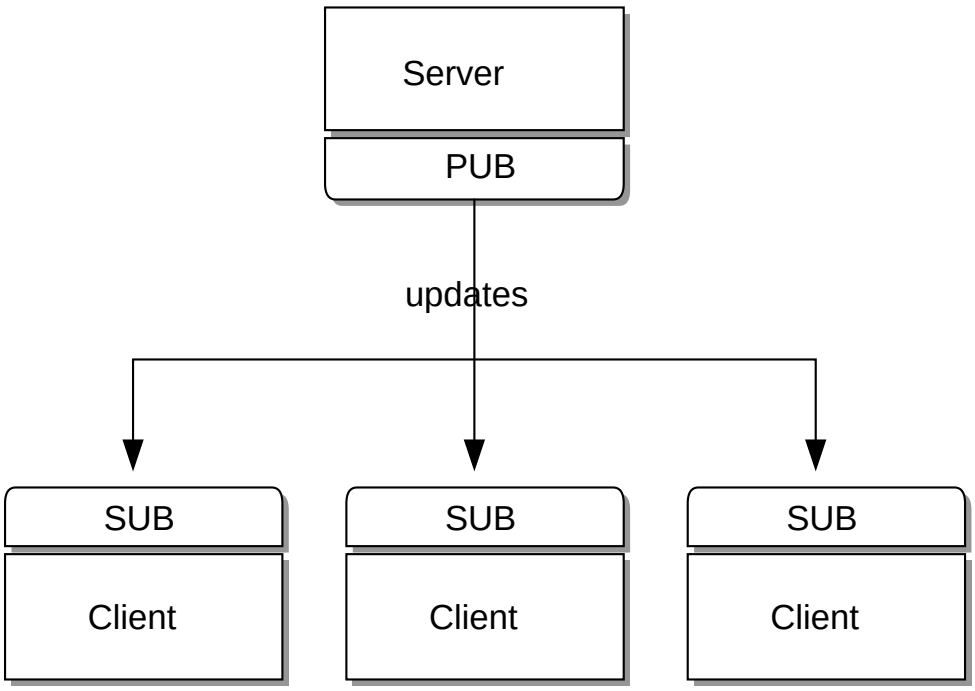
- Let a client join the network at any time, and reliably get the current server state.
- Let any client update the key-value cache (inserting new key-value pairs, updating existing ones, or deleting them).
- Reliably propagate changes to all clients, and do this with minimum latency overhead.
- Handle 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.

5.3.1. Distributing Key-Value Updates

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(Figure 5-3). We'll modify our client to store these in a hash table.

Figure 5-3. Simplest Clone Model



This is the server:

Example 5-2. Clone server, Model One (clonesrv1.lua)

```
(This example still needs translation into Lua)
```

And here is the client:

Example 5-3. Clone client, Model One (clonecli1.lua)

```
(This example still needs translation into Lua)
```

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 multi-part 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:

Example 5-4. Key-value message class (kvsimple.lua)

```
(This example still needs translation into Lua)
```

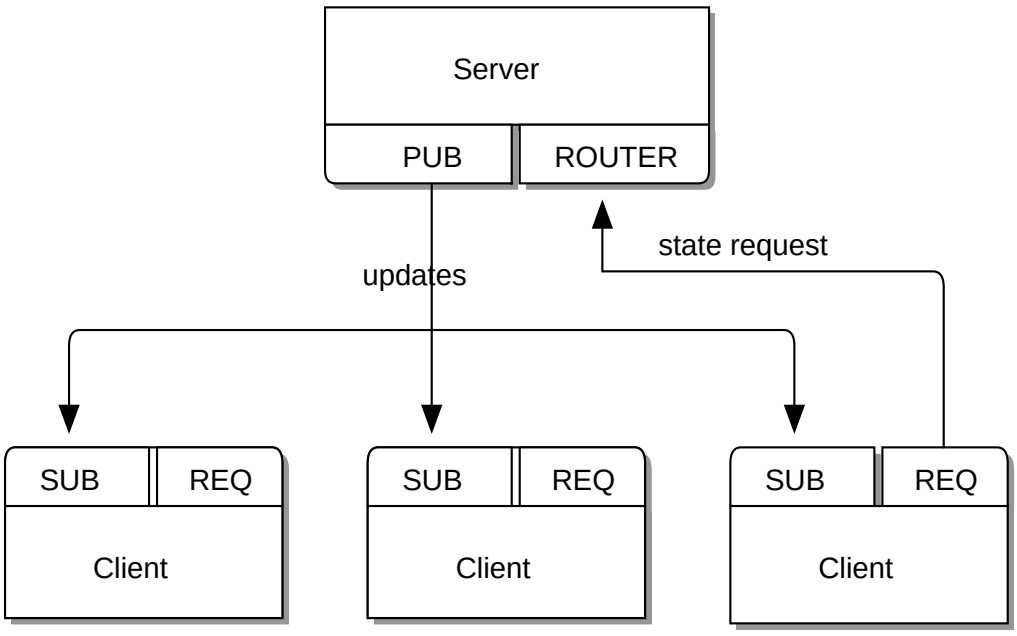
We'll make a more sophisticated kvmsg class later, for using in real applications.

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'.

5.3.2. Getting a Snapshot

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 5-4).

Figure 5-4. 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:

Example 5-5. Clone server, Model Two (clonesrv2.lua)

```
(This example still needs translation into Lua)
```

And here is the client:

Example 5-6. Clone client, Model Two (clonecli2.lua)

```
(This example still needs translation into Lua)
```

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.

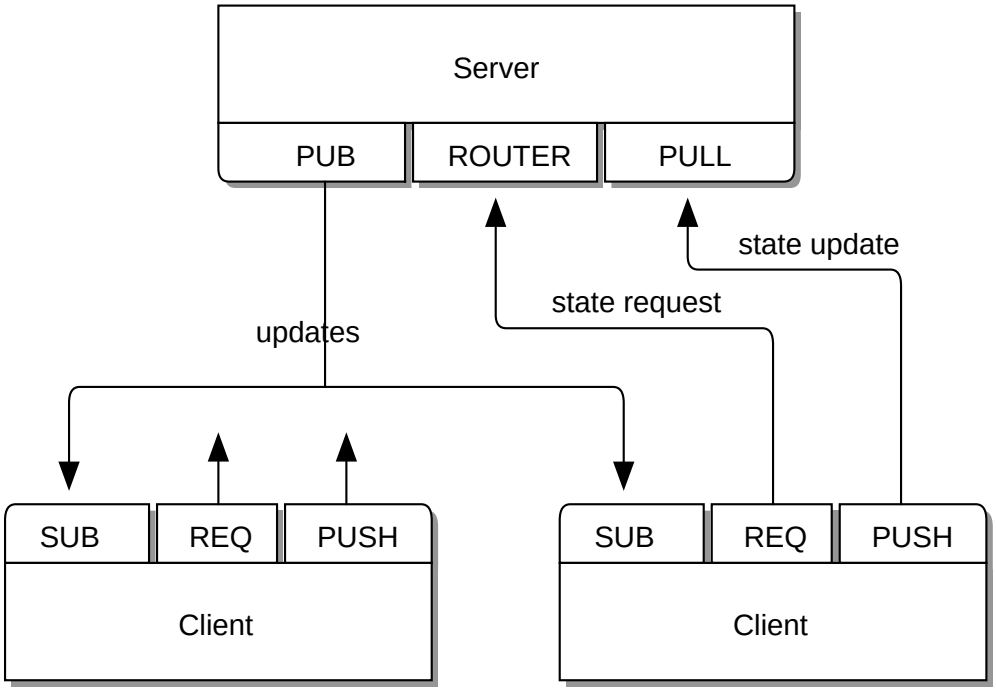
5.3.3. Republishing Updates

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 5-5).

Figure 5-5. 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:

Example 5-7. Clone server, Model Three (clonesrv3.lua)

```
(This example still needs translation into Lua)
```

And here is the client:

Example 5-8. Clone client, Model Three (clonecli3.lua)

```
(This example still needs translation into Lua)
```

Some notes about this code:

- The server has collapsed to a single task. 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 a real implementation we would drive updates from application code.

5.3.4. Clone Subtrees

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:

Example 5-9. Clone server, Model Four (clonesrv4.lua)

```
(This example still needs translation into Lua)
```

And here is the client:

Example 5-10. Clone client, Model Four (clonecli4.lua)

```
(This example still needs translation into Lua)
```

5.3.5. Ephemeral Values

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:

Example 5-11. Key-value message class - full (kvmsg.lua)

```
(This example still needs translation into Lua)
```

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:

Example 5-12. Clone server, Model Five (clonesrv5.lua)

```
(This example still needs translation into Lua)
```

5.3.6. Clone Server Reliability

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 (sips) 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.

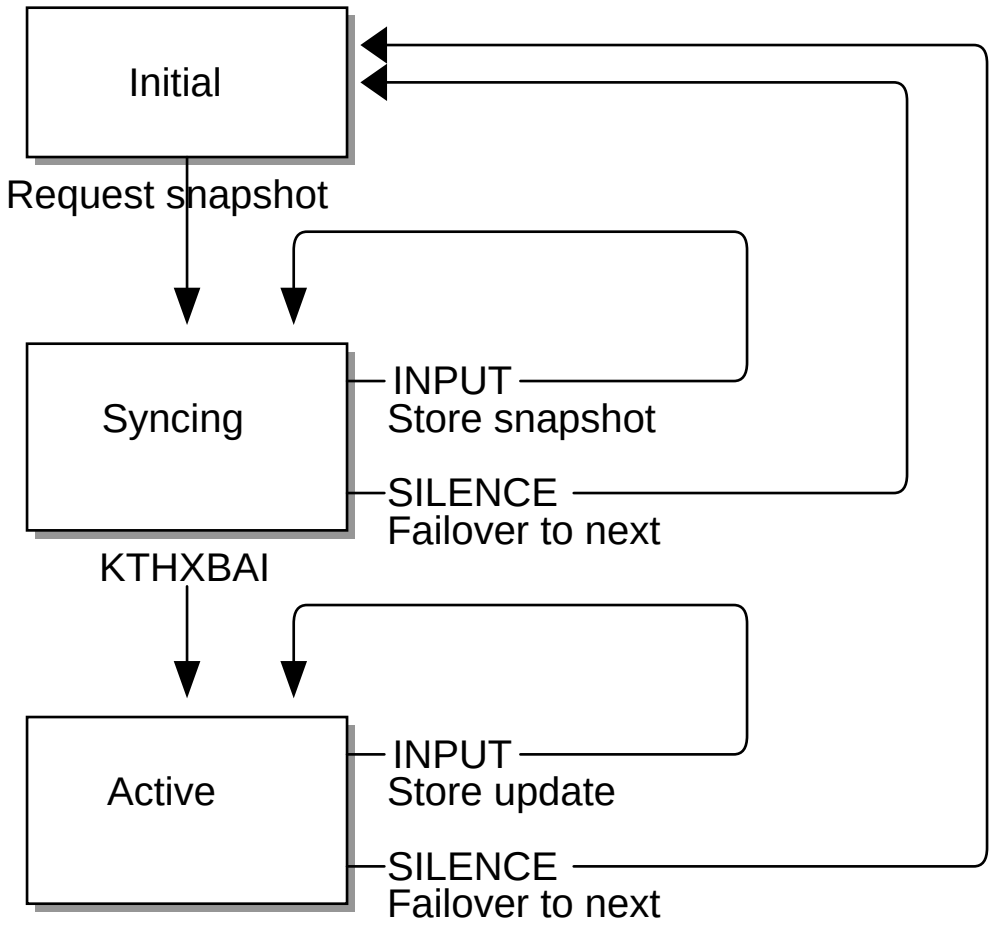
- We add heartbeats to server updates (to clients), so that a client can detect when the primary server has died. It can then switch over to the backup server.
- We connect the two servers using the Binary Star `bstar` reactor class. Binary Star relies on the clients to 'vote' by making an explicit request 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 state machine handles this(Figure 5-6), 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...)

Figure 5-6. Clone Client Finite State Machine



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.

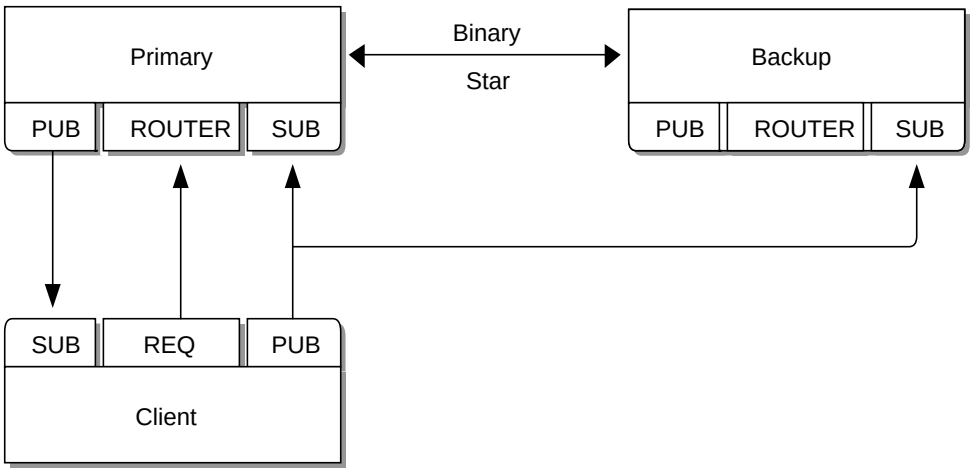
- Start to receive updates from clients, via its SUB socket.

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 the architecture for our high-availability server pair using the Binary Star pattern has two servers and a set of clients that talk to both servers(Figure 5-7).

Figure 5-7. 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 real-time, 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:

Example 5-13. Clone client, Model Six (clonecli6.lua)

```
(This example still needs translation into Lua)
```

And here is the actual clone class implementation:

Example 5-14. Clone class (clone.lua)

```
(This example still needs translation into Lua)
```

Finally, here is the sixth and last model of the clone server:

Example 5-15. Clone server, Model Six (clonesrv6.lua)

```
(This example still needs translation into Lua)
```

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 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 fail-over, 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.

5.3.7. Clone Protocol Specification

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

5.4. The Espresso Pattern

I'll end this chapter with a fun little machine that exploits the zmq_proxy[3] method to show you what's happening on a pub-sub network. It's deceptively simple:

Example 5-16. Espresso Machine (espresso.lua)

```
(This example still needs translation into Lua)
```

Chapter 6. The Human Scale

If you've survived the first five chapters, congratulations. It was hard for for me too. Happily the jokes and the code mostly write themselves, so we'll continue with our journey of exploring ØMQ. In this chapter I'm going to step back from the nuts and bolts of ØMQ's technical machinery, and look more at how to use ØMQ successfully in a larger project. Rather more opinion and experience, and a little less raw code.

We'll cover:

- What "software architecture" is really about.
- The Simplicity-Oriented Design process and its ugly cousins Cod and Tod.
- How to use ØMQ to go from idea to working prototype safely.
- Different ways to serialize your data as ØMQ messages.
- How to code-generate binary serialization codecs.
- How to build custom code generators.
- How to write and license an protocol specification.
- How to do fast restartable file transfer over ØMQ.
- How to do credit-based flow control.
- How to do heartbeating for different ØMQ patterns.
- How to build protocol servers and clients as state machines.
- How to make a secure protocol over ØMQ (yay!).
- A large-scale file publishing system (FileMQ).

6.1. The Tale of Two Bridges

Two old engineers were talking of their lives and boasting of their greatest projects. One of the engineers explained how he had designed one of the greatest bridges ever made.

"We built it across a river gorge," he told his friend. "It was wide and deep. We spent two years studying the land, and choosing designs and materials. We hired the best engineers and designed the bridge, which took another five years. We contracted the largest engineering firms to build the structures, the towers, the tollbooths, and the roads that would connect the bridge to the main highways. Dozens died during the construction. Under the road level we had trains, and a special path for cyclists. That bridge represented years of my life."

The second man reflected for a while, then spoke. "One evening me and a friend got drunk on vodka, and we threw a rope across a gorge," he said. "Just a rope, tied to two trees. There were two villages, one at

each side. At first, people pulled packages across that rope with a pulley and string. Then someone threw a second rope, and built a foot walk. It was dangerous, but the kids loved it. A group of men then rebuilt that, made it solid, and women started to cross, everyday, with their produce. A market grew up on one side of the bridge, and slowly that became a large town, since there was a lot of space for houses. The rope bridge got replaced with a wooden bridge, to allow horses and carts to cross. Then the town built a real stone bridge, with metal beams. Later, they replaced the stone part with steel, and today there's a suspension bridge standing in that same spot."

The first engineer was silent. "Funny thing," he said, "my bridge was demolished about ten years after we built it. Turns out it was built in the wrong place and no-one wanted to use it. Some guys had thrown a rope across the gorge, a few miles further downstream, and that's where everyone went."

6.2. Code on the Human Scale

To write a poem that captures the heart, first learn the language. To use ØMQ successfully at scale you have to learn two languages. The first is ØMQ itself. This takes even the best of us time. It's a truism that if you try to port an old architecture onto ØMQ, the results are going to be weird. ØMQ's language is subtle and profound and when you master it you will find yourself removing old complexity, not converting it.

However the real challenge of using ØMQ is that old barriers fall away, and the size of the projects you can do increases hugely. Non-distributed code is often a single-person project. You can work in your corner, perhaps for years, like an author on a book. It's all about concentration. But distributed code is different. To quote my favorite author, it "has to talk to code, has to be chatty, sociable, well-connected".

Writing distributed code is like playing live music: it's all about other people. Concentration is worthless if you can't listen. No-one enjoys listening to an amazingly proficient musician who's out of time with the rest of the group and can't read the mood of the audience. A live jam is entrancing not because of the technical quality but because of the real-time creative energy.

And so it goes with distributed code. Real-time creative energy is what wins, not pure technical quality, and certainly not technical quality combined with inability to work with others.

All this is fine in theory. Here comes the catch: working with other people is *plain hard*. We can expect a musician to be naturally social. But software developers? We're the very caricature of anti-social tunnel-visioned hermits. Other people are hard work. They're slow, they make mistakes, they ask too many questions, they don't respect our code, they make wrong assumptions, they argue.

My response isn't very sympathetic. To succeed in the software industry as it turns into something more like a never-ending live jam, we have to learn to put away our egos, work successfully with others, worry less about our own skills and look more at others, put away our natural insolence and attitude, and to learn to like and trust other people.

So this is what this chapter is really about: writing code at scale by understanding ourselves much better. Of course these lessons apply to all large-scale applications. Using ØMQ we just hit the problem sooner than we'd expect.

6.3. Psychology of Software Development

Dirkjan Ochtman pointed me to Wikipedia's definition of Software Architecture (http://en.wikipedia.org/wiki/Software_architecture) as "*the set of structures needed to reason about the system, which comprise software elements, relations among them, and properties of both*". For me this vapid and circular jargon is a good example of how miserably little we understand about what actually makes a successful large scale software architecture.

Architecture is the art and science of making large artificial structures for human use. If there is one thing I've learned and applied successfully in 30 years of making larger and larger software systems it is this: software is about people. Large structures in themselves are meaningless. It's how they function for *human use* that matters. And in software, human use starts with the programmers who make the software itself.

The core problems in software architecture are driven by human psychology, not technology. There are many ways our psychology affects our work. I could point to the way teams seem to get stupider as they get larger, or have to work across larger distances. Does that mean the smaller the team, the more effective? How then does a large global community like ØMQ manage to work successfully?

The ØMQ community wasn't accidental, it was a deliberate design, my contribution to the early days when the code came out of a cellar in Bratislava. The design was based on my pet science of "Social Architecture", which Wikipedia defines (http://en.wikipedia.org/wiki/Social_architecture) (what a coincidence!) as "*the process, and the product, of planning, designing, and growing an on-line community.*"

One of the tenets of Social Architecture is that *how we organize* is more significant than *who we are*. The same group, organized differently, can produce entirely opposite results. We are like peers in a ØMQ network, and our communication patterns have dramatic impact on our performance. Ordinary people, well connected, can far outperform a team of experts working in the wrong patterns. If you're the architect of a larger ØMQ application, you're going to have to help others find the right patterns for working together. Do this right, and your project can succeed. Do it wrong, and your project will fail.

The two most important psychological elements are IMO that we're really bad at understanding complexity, and that we are so good at working together to divide and conquer large problems. We're highly social apes, and kind of smart, but only in the right kind of crowd.

So here is my short list of the Psychological Elements of Software Architecture:

- **Stupidity**: our mental bandwidth is limited, so we're all stupid at some point. The architecture has to be simple to understand. This is the number one rule: simplicity beats functionality, every single time. If you can't understand an architecture on a cold gray Monday morning before coffee, it is too complex.
- **Selfishness**: we act only out of self-interest, so the architecture must create space and opportunity for selfish acts that benefit the whole. Selfishness is often indirect and subtle. For example I'll spend hours helping someone else understand something because that could be worth days to me later.
- **Laziness**: we make lots of assumptions, many of which are wrong. We are happiest when we can spend the least effort to get a result, to test an assumption quickly, so the architecture has to make this possible. Specifically, that means it must be simple.
- **Jealousy**: we're jealous of others, which means we'll overcome our stupidity and laziness to prove others wrong, and beat them in competition. The architecture thus has to create space for public competition based on fair rules that anyone can understand.
- **Reciprocity**: we'll pay extra in terms of hard work, even money, to punish cheats and enforce fair rules. The architecture should be heavily rule-based, telling people how to work together, but not what to work on.
- **Pride**: we're intensely aware of our social status, and we'll work hard to avoid looking stupid or incompetent in public. The architecture has to make sure every piece we make has our name on it, so we'll have sleepless nights stressing about what others will say about our work.
- **Greed**: we're ultimately economic animals (see selfishness), so the architecture has to give us economic incentive to invest in making it happen. Maybe it's polishing our reputation as experts, maybe it's literally making money from some skill or component. It doesn't matter what it is, but there must be economic incentive. Think of architecture as a market place, not an engineering design.
- **Conformity**: we're happiest to conform, out of fear and laziness, so the architecture should be strongly rule-based, and rules should be clear, accurate, well-documented, and enforced.
- **Fear**: we're unwilling to take risks, especially if it makes us look stupid. Fear of failure is a major reason people conform and follow the group in mass stupidity. The architecture should make silent experimentation easy and cheap, giving people opportunity for success without punishing failure.

These strategies work on large scale but also on small scale, within an organization or team.

6.4. The Bad, the Ugly, and the Delicious

Complexity is easy, it's simplicity that is hard. Whether our software is bad, ugly, or so delicious that it feels wrong to consume alone, doesn't depend so much on our individual skills as how we work together. That is, our processes.

There are many aspects to getting product-building teams and organizations to think wisely. You need diversity, freedom, challenge, resources, and so on. I discuss these in detail in Software and Silicon (http://swsi.info). However, even if you have all the right ingredients, the default processes that skilled engineers and designers develop will result in complex, hard-to-use products.

The classic errors are: to focus on ideas, not problems; to focus on the wrong problems; to misjudge the value of solving problems; to not use ones' own work; and in many other ways to misjudge the real market.

I'll propose a process called "Simplicity Oriented Design", or SOD, which is as far as I can tell a reliable, repeatable way of developing simple and elegant products. This process organizes people into flexible supply chains that are able to navigate a problem landscape rapidly and cheaply. They do this by building, testing, and keeping or discarding minimal plausible solutions, called "patches". Living products consist of long series of patches, applied one atop the other. Yes, you may recognize the process by which we develop ØMQ.

Let's first look at the more common and less joyful processes, TOD and COD.

6.4.1. Trash-Oriented Design

The most popular design process in large businesses seems to be "Trash Oriented Design", or TOD. TOD feeds off the belief that all we need to make money are great ideas. It's tenacious nonsense but a powerful crutch for people who lack imagination. The theory goes that ideas are rare, so the trick is to capture them. It's like non-musicians being awed by a guitar player, not realizing that great talent is so cheap it literally plays on the streets for coins.

The main output of TODs are expensive "ideations": concepts, design documents, and products that go straight into the trash can. It works as follows:

- The Creative People come up with long lists of "we could do X and Y". I've seen endlessly detailed lists of everything amazing a product could do. Once the creative work of idea generation has happened, it's just a matter of execution, of course.
- So the managers and their consultants pass their brilliant, world-shattering ideas to designers who acres of detailed, preciously refined design documents. The designers take the tens of ideas the managers came up with, and turn them into hundreds of amazing, world-changing designs.
- These designs get given to engineers who scratch their heads and wonder who the heck came up with such stupid nonsense. They start to argue back but the designs come from up high, and really, it's not up to engineers to argue with creative people and expensive consultants.
- So the engineers creep back to their cubicles, humiliated and threatened into building the gigantic but oh-so-elegant pile of junk. It is bone-breakingly hard work since the designs take no account of practical costs. Minor whims might take weeks of work to build. As the project gets delayed, the managers bully the engineers into giving up their evenings and weekends.
- Eventually, something resembling a working product makes it out of the door. It's creaky and fragile, complex and ugly. The designers curse the engineers for their incompetence and pay more consultants to put lipstick onto the pig, and slowly the product starts to look a little nicer.
- By this time, the managers have started to try to sell the product and they find, shockingly, that no-one wants it. Undaunted and courageously they build million-dollar web sites and ad campaigns to explain

to the public why they absolutely need this product. They do deals with other businesses to force the product on the lazy, stupid and ungrateful market.

- After twelve months of intense marketing, the product still isn't making profits. Worse, it suffers dramatic failures and gets branded in the press as a disaster. The company quietly shelves it, fires the consultants, buys a competing product from a small start-up and re-brands that as its own Version 2. Hundreds of millions of dollars end-up in the trash.
- Meanwhile, another visionary manager, somewhere in the Organization, drinks a little too much tequila with some marketing people and has a Brilliant Idea.

Trash-Oriented Design would be a caricature if it wasn't so common. Something like 19 out of 20 market-ready products built by large firms are failures (yes, 87% of statistics are made up on the spot). The remaining one in 20 probably only succeeds because the competitors are so bad and the marketing is so aggressive.

The main lessons of TOD are quite straight-forward but hard to swallow. They are:

- Ideas are cheap. No exceptions. There are no brilliant ideas. Anyone who tries to start a discussion with "oooh, we can do this too!" should be beaten down with all the passion one reserves for traveling evangelists. It is like sitting in a cafe at the foot of a mountain, drinking a hot chocolate and telling others, "*hey, I have a great idea, we can climb that mountain! And build a chalet on top! With two saunas! And a garden! Hey, and we can make it solar powered! Dude, that's awesome! What color should we paint it? Green! No, blue! OK, go and make it, I'll stay here and make spreadsheets and graphics!*"
- The starting point for a good design process is to collect real problems that confront real people. The second step is to evaluate these problems with the basic question, "how much is it worth to solve this problem?" Having done that, we can collect that set of problems that are worth solving.
- Good solutions to real problems will succeed as products. Their success will depend on how good and cheap the solution is, and how important the problem is (and sadly, how big the marketing budgets are). But their success will also depend on how much they demand in effort to use, in other words how simple they are.

Hence after slaying the dragon of utter irrelevance, we attack the demon of complexity.

6.4.2. Complexity-Oriented Design

Really good engineering teams and small firms can usually build decent products. But the vast majority of products still end up being too complex and less successful than they might be. This is because specialist teams, even the best, often stubbornly apply a process I call "Complexity-Oriented Design", or COD, which works as follows:

- Management correctly identifies some interesting and difficult problem with economic value. In doing so they already leapfrog over any TOD team.

- The team with enthusiasm start to build prototypes and core layers. These work as designed and thus encouraged, the team go off into intense design and architecture discussions, coming up with elegant schemas that look beautiful and solid.
- Management comes back and challenges team with yet more difficult problems. We tend to equate value with cost, so the harder the problem, and more expensive to solve, the more the solution should be worth, in their minds.
- The team, being engineers and thus loving to build stuff, build stuff. They build and build and build and end-up with massive, perfectly-designed complexity.
- The products go to market, and the market scratches its head and asks, "seriously, is this the best you can do?" People do use the products, especially if they aren't spending their own money in climbing the learning curve.
- Management gets positive feedback from its larger customers, who share the same idea that high cost (in training and use) means high value. and so continues to push the process.
- Meanwhile somewhere across the world, a small team is solving the same problem using a better process, and a year later smashes the market to little pieces.

COD is characterized by a team obsessively solving the wrong problems to the point of collective insanity. COD products tend to be large, ambitious, complex, and unpopular. Much open source software is the output of COD processes. It is insanely hard for engineers to **stop** extending a design to cover more potential problems. They argue, "what if someone wants to do X?" but never ask themselves, "what is the real value of solving X?"

A good example of COD in practice is Bluetooth, a complex, over-designed set of protocols that users hate. It continues to exist only because in a massively-patented industry there are no real alternatives. Bluetooth is perfectly secure, which is close to pointless for a proximity protocol. At the same time it lacks a standard API for developers, meaning it's really costly to use Bluetooth in applications.

On the #zeromq IRC channel, Wintre once wrote of how enraged he was many years ago when he "*found that XMMS 2 had a working plugin system but could not actually play music*."

COD is a form of large-scale "rabbit holing", in which designers and engineers cannot distance themselves from the technical details of their work. They add more and more features, utterly misreading the economics of their work.

The main lessons of COD are also simple but hard for experts to swallow. They are:

- Making stuff that you don't immediately have a need for is pointless. Doesn't matter how talented or brilliant you are, if you just sit down and make stuff people are not actually asking for, you are most likely wasting your time.
- Problems are not equal. Some are simple, and some are complex. Ironically, solving the simpler problems often has more value to more people than solving the really hard ones. So if you allow engineers to just work on random things, they'll most focus on the most interesting but least worthwhile things.

- Engineers and designers love to make stuff and decoration, and this inevitably leads to complexity. It is crucial to have a "stop mechanism", a way to set short, hard deadlines that force people to make smaller, simpler answers to just the most crucial problems.

6.4.3. Simplicity-Oriented Design

Finally, we come to the rare but precious Simplicity-Oriented Design. This process starts with a realization: we do not know what we have to make until after we start making it. Coming up with ideas, or large-scale designs isn't just wasteful, it's a direct hindrance to designing the truly accurate solutions. The really juicy problems are hidden like far valleys, and any activity except active scouting creates a fog that hides those distant valleys. You need to keep mobile, pack light, and move fast.

SOD works as follows:

- We collect a set of interesting problems (by looking at how people use technology or other products) and we line these up from simple to complex, looking for and identifying patterns of use.
- We take the simplest, most dramatic problem and we solve this with a minimal plausible solution, or "patch". Each patch solves exactly a genuine and agreed problem in a brutally minimal fashion.
- We apply one measure of quality to patches, namely "can this be done any simpler while still solving the stated problem?" We can measure complexity in terms of concepts and models that the user has to learn or guess in order to use the patch. The fewer, the better. A perfect patch solves a problem with zero learning required by the user.
- Our product development consists of a patch that solves the problem "we need a proof of concept" and then evolves in an unbroken line to a mature series of products, through hundreds or thousands of patches piled on top of each other.
- We do not do *anything* that is not a patch. We enforce this rule with formal processes that demand that every activity or task is tied to a genuine and agreed problem, explicitly enunciated and documented.
- We build our projects into a supply chain where each project can provide problems to its "suppliers" and receive patches in return. The supply chain creates the "stop mechanism" since when people are impatiently waiting for an answer, we necessarily cut our work short.
- Individuals are free to work on any projects, and provide patches at any place they feel it's worthwhile. No individuals "own" any project, except to enforce the formal processes. A single project can have many variations, each a collection of different, competing patches.
- Projects export formal and documented interfaces so that upstream (client) projects are unaware of change happening in supplier projects. Thus multiple supplier projects can compete for client projects, in effect creating a free and competitive market.
- We tie our supply chain to real users and external clients and we drive the whole process by rapid cycles so that a problem received from outside users can be analyzed, evaluated, and solved with a patch in a few hours.
- At every moment from the very first patch, our product is shippable. This is essential, because a large proportion of patches will be wrong (10-30%) and only by giving the product to users can we know which patches have become problems and themselves need solving.

SOD is a form of "hill climbing algorithm", a reliable way of finding optimal solutions to the most significant problems in an unknown landscape. You don't need to be a genius to use SOD successfully, you just need to be able to see the difference between the fog of activity and the progress towards new real problems.

A really good designer with a good team can use SOD to build world-class products, rapidly and accurately. To get the most out of SOD, the designer has to use the product continuously, from day 1, and develop his or her ability to smell out problems such as inconsistency, surprising behavior, and other forms of friction. We naturally overlook many annoyances but a good designer picks these up, and thinks about how to patch them. Design is about removing friction in the use of a product.

In an open source setting, we do this work in public. There's no "let's open the code" moment. Projects that do this are in my view missing the point of open source, which is to engage your users in your exploration, and to build community around the seed of the architecture.

6.5. Message Oriented Pattern for Elastic Design

Now I'll introduce MOPED, which is a SOD pattern custom-designed for ØMQ architectures. It was either MOPED or BIKE, the Backronym-Induced Kinetic Effect. That's short for BICICLE, the Backronym-Inflated See if I Care Less Effect. In life, one learns to go with the least embarrassing choices.

Speaking of embarrassments, just as ØMQ lets us aim for really massive architectures, it also, like any technology that removes friction, opens the door to truly massive blunders. If ØMQ is the ACME rocket-propelled shoe of distributed software development, a lot of us are like Wile E. Coyote, slamming full speed into the proverbial desert cliff.

So MOPED is meant to save us from such mistakes. Partly it's about slowing down, partly it's about ensuring that when you move fast, you go - and this is essential, dear reader - in the *right direction*. It's my standard interview riddle: what's the rarest property of any software system, the absolute hardest thing to get right, the lack of which causes the slow or fast death of the vast majority of projects? The answer is not code quality, funding, performance, or even (though it's a close answer), popularity. The answer is "accuracy".

If you've read the Guide observantly you'll have seen MOPED in action already. The development of Majordomo in Chapter 4 is a near-perfect case. But cute names are worth a thousand words.

The goal of MOPED is to define a process, a pattern by which we can take a rough use case for a new distributed application, and go from "hello world" to fully-working prototype in any language in under a week.

Using MOPED, you grow, more than build, a working ØMQ architecture from the ground-up, with minimal risk of failure. By focusing on the contracts, rather than the implementations, you avoid the risk of premature optimization. By driving the design process through ultra-short test-based cycles, you can be more certain what you have works, before you add more.

We can turn this into five real steps:

- Step 1: internalize the ØMQ semantics.
- Step 2: draw a rough architecture.
- Step 3: decide on the contracts.
- Step 4: make a minimal end-to-end solution.
- Step 5: solve one problem and repeat.

6.5.1. Step 1: Internalize the Semantics

To repeat myself: you must learn ØMQ's language. The only way to learn a language is to use it. There's no way to avoid this investment, no tapes you can play while you sleep, no chips you can plug in to magically become smarter. Read the Guide, work through the code examples, understand what's going on, and (most importantly) write some examples yourself, and then *throw them away*.

At a certain point you'll feel a clicking noise in your brain. Maybe you'll have a weird chili-induced dream where little ØMQ tasks run around trying to eat you alive. Maybe you'll just think "aaahh, so *that's* what it means!" If we did our work right, it should take 2-3 days. However long it takes, until you start thinking in terms of ØMQ sockets and patterns, you're not ready for step 2.

6.5.2. Step 2: Draw a Rough Architecture

Whiteboard time. Get a couple of colleagues and try to draw your architecture on a whiteboard. you want to draw boxes connected with arrows, showing the flow of work, data, results, etc. Since we live in a gravity well, it's best to draw the main arrows going down. Almost all architectures have a *direction*, and a certain symmetry, and what you want to do is capture that as simply and cleanly as you can.

Ignore anything that's not central to the core problem. Ignore logging, error handling, recovery from failures, etc. What you leave out is as important as what you capture: you can always add, but it's very hard to remove. When you have a simple, clean drawing, you're ready for step 3.

6.5.3. Step 3: Decide on the Contracts

Human scale depends on contracts, and the more explicit they are, the better things scale. You don't care *how* things happen, only the results. If I send an email, I don't care how it arrives at its destination, so

long as the contract (it arrives within a few minutes, it's not modified, it doesn't get lost) is respected.

And to build a large system that works well, you must focus on the contracts, before the implementations. It may sound obvious but all too often, people forget and ignore this, or are just too shy to impose themselves. I wish I could say ØMQ had done this properly but for years our public contracts were second-rate afterthoughts instead of primary in-your-face pieces of work.

So what is a contract in a distributed system? There are, in my experience, two types of contract:

- The APIs to client applications. Remember the Psychological Elements. The APIs need to be as absolutely *simple*, *consistent*, and *familiar* as possible. Yes, you can generate API documentation from code, but you must first design it, and designing an API is often hard.
- The protocols that connect the pieces. It sounds like rocket science, but it's really just a simple trick, and one that ØMQ makes particularly easy. In fact they're so simple to write, and need so little bureaucracy that I call them "unprotocols".

You write minimal contracts that are mostly just place markers. Most messages and most API methods will be missing, or empty. You also want to write down any known technical requirements in terms of throughput, latency, reliability, etc. These are the criteria on which you will accept, or reject, any particular piece of work.

6.5.4. Step 4: Write a Minimal End-to-End Solution

The goal is to test out the overall architecture as rapidly as possible. Make skeleton applications that call the APIs, and skeleton stacks that implement both sides of every protocol. You want to get a working end-to-end "hello world" as soon as you can. You want to be able to test code, as you write it, to weed-out the broken assumptions and inevitable errors you make. Do not go off and spend six months writing a test suite! Instead, make a minimal bare-bones application that uses our still-hypothetical API.

If you design an API wearing the hat of the person who implements it, you'll start to think of performance, features, options, and so on. You'll make it more complex, more irregular, and more surprising than it should be. But, and here's the trick (it's a cheap one, was big in Japan), if you design an API while wearing the hat of the poor sucker who has to actually write apps that use it, you use all that laziness and fear to our advantage.

Write down the protocols, on a wiki or shared document, in such a way that you can explain every command clearly without too much detail. Strip off any real functionality, because it'll create inertia that just makes it harder to move stuff around. You can always add weight. Don't spend effort defining formal message structures: pass the minimum around, in the simplest possible fashion, using ØMQ's multi-part framing.

Our goal is to get the simplest test case working, without any avoidable functionality. Everything you can chop off the list of things to do, you chop. Ignore the groans from colleagues and bosses. I'll repeat this

once again: you can *always* add functionality, that's relatively easy. But aim to keep the overall weight to a minimum.

6.5.5. Step 5: Solve One Problem and Repeat

You're now in the Happy Loop of issue-driven development where you can start to solve tangible problems instead of adding features. Write issues that state a clear problem, and propose a solution. Keep in mind, as you design the API, your standards for names, consistency, and behavior. Writing these down in prose often helps keep them sane.

From here, every single change you make to the architecture and code is now proven by running the test case, watching it not work, making the change, and then watching it work.

Now you go through the whole cycle (extending the test case, fixing the API, updating the protocol, extending the code, as needed), taking problems one at a time and testing the solutions individually. It should take about 10-30 minutes for each cycle, with the occasional spike due to random confusion.

6.6. Unprotocols

6.6.1. Why Unprotocols?

When this man thinks of protocols, this man thinks of massive documents written by committees, over years. This man thinks of the IETF, W3C, ISO, Oasis, regulatory capture, FRAND patent license disputes, and soon after, this man thinks of retirement to a nice little farm in northern Bolivia up in the mountains where the only other needlessly stubborn beings are the goats chewing up the coffee plants.

Now, I've nothing personal against committees. The useless folk need a place to sit out their lives with minimal risk of reproducing, after all, that only seems fair. But most committee protocols tend towards complexity (the ones that work), or trash (the ones we don't talk about). There's a few reasons for this. One is the amount of money at stake. More money means more people who want their particular prejudices and assumptions expressed in prose. But two is the lack of good abstractions on which to build. People have tried to build reusable protocol abstractions, like BEEP. Most did not stick, and those that did, like SOAP and XMPP, are on the complex side of things.

It used to be, decades ago, when the Internet was a young modest thing, that protocols were short and sweet. They weren't even "standards", but "requests for comments", which is as modest as you can get. It's been one of my goals since we started iMatix in 1995 to find a way for ordinary people like me to write small, accurate protocols without the overhead of the committees.

Now, ØMQ does appear to provide a living, successful protocol abstraction layer with its "we'll carry multi-part messages over random transports" way of working. Since ØMQ deals silently with framing, connections, and routing, it's surprisingly easy to write full protocol specs on top of ØMQ, and in Chapters four and five I showed how to do this.

Somewhere around mid-2007, I kicked-off the Digital Standards Organization to define new simpler ways of producing little standards, protocols, specifications. In my defense, it was a quiet summer. At the time I wrote that (http://www.digistan.org/spec:1) a new specification should take "*minutes to explain, hours to design, days to write, weeks to prove, months to become mature, and years to replace.*"

In 2010 we started calling such little specifications "unprotocols", which some people might mistake for a dastardly plan for world domination by a shadowy international organization, but which really just means, "protocols without the goats".

6.6.2. How to Write Unprotocols

Here's an unprotocol called NOM that we'll come back to later in this chapter:

```
nom-protocol    = open-peering *use-peering

open-peering    = C:OHAI ( S:OHAI-OK / S:WTF )

use-peering     = C:ICANHAZ
                / S:CHEEZBURGER
                / C:HUGZ S:HUGZ-OK
                / S:HUGZ C:HUGZ-OK
```

I've actually used these keywords (OHAI, WTF) in commercial projects. They make developers giggly and happy. They confuse management. They're good in first drafts that you want to throw away later.

When you start to write unprotocols, stick to a consistent structure so that your readers know what to expect. Here is the structure I use:

- Cover section: with a 1-line summary, URL to the spec, formal name, version, who to blame.
- License for the text: absolutely needed for public specifications.
- The change process: i.e. how I as a reader fix problems in the specification?
- Use of language: MUST, MAY, SHOULD, etc. with a reference to RFC 2119.
- Maturity indicator: is this a experimental, draft, stable, legacy, retired?
- Goals of the protocol: what problems is it trying to solve?
- Formal grammar: prevents arguments due to different interpretation of the text.
- Technical explanation: semantics of each message, error handling, etc.
- Security discussion: explicitly, how secure the protocol is.

- References: to other documents, protocols, etc.

Writing clear, expressive text is hard. Do avoid trying to describe implementations of the protocol. Remember that you're writing a contract. You describe in clear language the obligations and expectations of each party, the level of obligation, and the penalties for breaking the rules. You do not try to define *how* each party honors its part of the deal.

If you need reference material to start with, read the http://rfc.zeromq.org site, which has a bunch of unprotocols that you can copy/paste from.

Here are some key points about unprotocols:

- As long as your process is open then you don't need a committee: just make clean minimal designs and make sure anyone is free to improve them.
- If use an existing license then you don't have legal worries afterwards. I use GPLv3 for my public specifications and advise you to do the same. For in-house work, standard copyright is perfect.
- The formality is valuable. That is, learn to write ABNF (http://www.ietf.org/rfc/rfc2234.txt) and use this to fully document your messages.
- Use a market-driven life-cycle process like Digistan's COSS (http://www.digistan.org/spec:1) so that people place the right weight on your specs as they mature (or don't).

6.6.3. Why use the GPLv3 for Public Specifications?

The license you choose is particularly crucial for public specifications. Traditionally, protocols are published under custom licenses, where the authors own the text and derived works are forbidden. This sounds great (after all, who wants to see a protocol forked?) but it's in fact highly risky. A protocol committee is vulnerable to capture, and if the protocol is important and valuable, the incentive for capture grows.

Once captured, like some wild animals, an important protocol will often die. The real problem is there's no way to *free* a captive protocol published under a conventional license. The word "free" isn't just an adjective to describe speech or air, it's also a verb, and the right to fork a work, *against the wishes of the owner*, is essential to avoiding capture.

Let me explain this in shorter words. Imagine iMatix writes a protocol today, that's really amazing and popular. We publish the spec and many people implement it. Those implementations are fast and awesome, and free as in beer. And they start to threaten an existing business. Their expensive commercial product is slower and can't compete. So one day they come to our iMatix office in Maetang-Dong, South Korea, and offer to buy our firm. Since we're spending vast amounts on sushi and beer and GFEs, we accept gratefully. With evil laughter the new owners of the protocol stop improving the public version, and close the specification and add patented extensions. Their new products support this, and they take over the whole market.

When you contribute to an open source project, you really want to know your hard work won't used against you by a closed-source competitor. Which is why the GPL beats the "more permissive" BSD/MIT/X11 licenses. These license give permission to cheat. This applies just as much to protocols as to source code.

When you implement a GPLv3 specification, your applications are of course yours, and licensed any way you like. But you can be sure and certain of two things. One, that specification will *ever* be embraced and extended into proprietary forms. Any derived forms of the specification must also be GPLv3. Two, no-one who ever implements or uses the protocol will ever launch a patent attack on anything it covers.

6.7. Serializing your Data

When we start to design a protocol, one of the first questions we face is how we encode data on the wire. There is, sadly, no universal answer. There are a half-dozen different ways to serialize data, each with pros and cons. We'll explore these.

However, there is a general lesson I've learned over a couple of decades of writing protocols small and large. I call this the "Cheap and Nasty" pattern: you can often split your work into two layers, and solve these separately, one using a "cheap" approach, the other using a "nasty" approach.

6.7.1. Cheap and Nasty

The key insight to making Cheap and Nasty work is to realize that many protocols mix a low-volume chatty part for control, and a high-volume asynchronous part for data. For instance, HTTP has a chatty dialog to authenticate and get pages, and an asynchronous dialog to stream data. FTP actually splits this over two ports; one port for control and one port for data.

Protocol designers who don't separate control from data tend to make awful protocols, because the trade-offs in the two cases are almost totally opposite. What is perfect for control is terrible for data, and what's ideal for data just doesn't work for control. It's especially true when we want high-performance at the same time as extensibility and good error checking.

Let's break this down using a classic client-server use-case. The client connects to the server, and authenticates. It then asks for some resource. The server chats back, then starts to send data back to the client. Eventually the client disconnects or the server finishes, and the conversation is over.

Now, before starting to design these messages, stop and think, and let's compare the control dialog, and the data flow:

- The control dialog lasts a short time and involve very few messages. The data flow could last for hours or days, and involve billions of messages.

- The control dialog is where all the "normal" errors happen, e.g. not authenticated, not found, payment required, censored, etc. Any errors that happen during the data flow are exceptional (disk full, server crashed).
- The control dialog is where things will change over time, as we add more options, parameters, and so on. The data flow should barely change over time since the semantics of a resource are fairly constant over time.
- The control dialog is essentially a synchronous request/reply dialog. The data flow is essentially a 1-way asynchronous flow.

These differences are critical. When we talk about performance, it applies *only* to data flows. It's pathological to design a one-time control dialog to be fast. When we talk about the cost of serialization, thus, this only applies to the data flow. The cost of encoding/decoding the control flow could be huge, and for many cases it would not change a thing. So, we encode control using "Cheap", and we encode data flows using "Nasty".

Cheap is essentially synchronous, verbose, descriptive, and flexible. A Cheap message is full of rich information that can change for each application. Your goal as designer is to make this information easy to encode and to parse, trivial to extend for experimentation or growth, and highly robust against change both forwards and backwards. The Cheap part of a protocol looks like this:

- It uses a simple self-describing structured encoding for data, be it XML, JSON, HTTP-style headers, or some other. Any encoding is fine so long as there are standard simple parsers for it in your target languages.
- It uses a straight request-reply model where each request has a success/failure reply. This makes it trivial to write correct clients and servers for a Cheap dialog.
- It doesn't try, even marginally, to be fast. Performance doesn't matter when you do something once or a few times per session.

A Cheap parser is something you take off the shelf, and throw data at. It shouldn't crash, shouldn't leak memory, should be highly tolerant, and should be relatively simple to work with. That's it.

Nasty however is essentially asynchronous, terse, silent, and inflexible. A Nasty message carries minimal information that practically never changes. Your goal as designer is to make this information ultrafast to parse, and possibly even impossible to extend and experiment with. The ideal Nasty pattern looks like this:

- It uses a hand-optimized binary layout for data, where every bit is precisely crafted.
- It uses a pure asynchronous model where one or both peers send data without acknowledgments (or if they do, they use sneaky asynchronous techniques like credit-based flow control).
- It doesn't try, even marginally, to be friendly. Performance is all that matters when you are doing something several million times per second.

A Nasty parser is something you write by hand, which writes or reads bits, bytes, words, and integers individually and precisely. It rejects anything it doesn't like, does no memory allocations at all, and never

crashes.

Cheap and Nasty isn't a universal pattern; not all protocols have this dichotomy. Also, how you use Cheap and Nasty will depend. In some cases, it can be two parts of a single protocol. In other cases it can be two protocols, one layered on top of the other.

6.7.2. ØMQ Framing

The simplest and most widely used serialization format for ØMQ applications is ØMQ's own multi-part framing. For example, here is how the Majordomo Protocol (http://rfc.zeromq.org/spec:7) defines a request:

```
Frame 0: Empty frame
Frame 1: "MDPW01" (six bytes, representing MDP/Worker v0.1)
Frame 2: 0x02 (one byte, representing REQUEST)
Frame 3: Client address (envelope stack)
Frame 4: Empty (zero bytes, envelope delimiter)
Frames 5+: Request body (opaque binary)
```

To read and write this in code is easy. But this is a classic example of a control flow (the whole of MDP is, really, since it's a chatty request-reply protocol). When we came to improve MDP for the second version, we had to change this framing. Excellent, we broke all existing implementations!

Backwards compatibility is hard, but using ØMQ framing for control flows *does not help*. Here's how I should have designed this protocol if I'd followed by own advice (and I'll fix this in the next version). It's split into a Cheap part and a Nasty part, and uses the ØMQ framing to separate these:

```
Frame 0: "MDP/2.0" for protocol name and version
Frame 1: command header
Frame 2: command body
```

Where we'd expect the parse the command header in the various intermediaries (client API, broker, and worker API), and pass the command body untouched from application to application.

6.7.3. Serialization Languages

Serialization languages have their fashions. XML used to be big as in popular, then it got big as in over-engineered, and then it fell into the hands of "Enterprise Information Architects" and it's not been seen alive since. Today's XML is the epitome of "somewhere in that mess is small, elegant language trying to escape".

Still XML, was way, way better than its predecessors which included such monsters as the Standard Generalized Markup Language (SGML), which in turn were a cool breeze compared to mind-torturing

beasts like EDIFACT. So the history of serialization languages seems to be of gradually emerging sanity, hidden by waves of revolting EIAs doing their best to hold onto their jobs.

JSON popped out of the JavaScript world as a quick-and-dirty "I'd rather resign than use XML here" way to throw data onto the wire and get it back again. JSON is just minimal XML expressed, sneakily, as JavaScript source code.

Here's a simple example of using JSON in a Cheap protocol:

```
"protocol": {
    "name": "MTL",
    "version": 1
},
"virtual-host": "test-env"
```

The same in XML would be (XML forces us to invent a single top-level entity):

```
<command>
    <protocol name = "MTL" version = "1" />
    <virtual-host>test-env</virtual-host>
</command>
```

And using plain-old HTTP-style headers:

```
Protocol: MTL/1.0
Virtual-host: test-env
```

These are all pretty equivalent so long as you don't go overboard with validating parsers, schemas and such "trust us, this is all for your own good" nonsense. A Cheap serialization language gives you space for experimentation for free ("ignore any elements/attributes/headers that you don't recognize"), and it's simple to write generic parsers that e.g. thunk a command into a hash table, or vice-versa.

However it's not all roses. While modern scripting languages support JSON and XML easily enough, older languages do not. If you use XML or JSON, you create non-trivial dependencies. It's also somewhat of a pain to work with tree-structured data in a language like C.

So you can drive your choice according to the languages you're aiming for. If your universe is a scripting language then go for JSON. If you are aiming to build protocols for wider system use, keep things simple for C developers and stick to HTTP-style headers.

6.7.4. Serialization Libraries

The msgpack.org site says, "*It's like JSON. but fast and small. MessagePack is an efficient binary serialization format. It lets you exchange data among multiple languages like JSON but it's faster and*

smaller. For example, small integers (like flags or error code) are encoded into a single byte, and typical short strings only require an extra byte in addition to the strings themselves."

I'm going to make the perhaps unpopular claim that "fast and small" are features that solve non-problems. The only real problem that serialization libraries solve is, as far as I can tell, the need to document the message contracts and actually serialize data to and from the wire.

Let's start with "fast and small". It's based on a two-part argument. First, that making your messages smaller, and that reducing CPU cost for encoding and decoding will make a significant different to your application's performance. Second, that this equally valid across-the-board to all messages.

But most real applications tend to fall into one of two categories. Either the speed of serialization and size of encoding is marginal compared to other costs, such as database access or application code performance. Or, network performance really is critical, and then all significant costs occur in a few specific message types.

Thus, aiming for "fast and small" across the board is a false optimization. You neither get the easy flexibility of Cheap for your infrequent control flows, nor do you get the brutal efficiency of Nasty for your high-volume data flows. Worse, the assumption that all messages are equal in some way can corrupt your protocol design. Cheap and Nasty isn't only about serialization strategies, it's also about synchronous vs. asynchronous, error handling, and the cost of change.

My experience is that most performance problems in message-based applications can be solved by (a) improving the application itself and (b) hand-optimizing the high-volume data flows. And to hand-optimize your most critical data flows, you need to cheat, know and exploit facts about your data, which is something general-purpose serializers cannot do.

Now to documentation: the need to write our contracts explicitly and formally, not in code. This is a valid problem to solve, indeed one of the main ones if we're to build a long-lasting large-scale message-based architecture.

Here is how we describe a typical message using the MessagePack IDL:

```
message Person {
  1: string surname
  2: string firstname
  3: optional string email
}
```

Now, the same message using the protobufs IDL:

```
message Person {
  required string surname = 1;
  required string firstname = 2;
  optional string email = 3;
```

```
}
```

It works but in most practical cases, wins you little over a serialization language backed by decent specifications written by hand or produced mechanically (we'll come to this). The price you'll pay is an extra dependency, and quite probably, worse overall performance than if you used Cheap and Nasty.

6.7.5. Hand-written Binary Serialization

As you'll gather from this book, my preferred language for systems programming is C (upgraded to C99, with a constructor/destructor API model and generic containers). There are two reasons I like this modernized C language: firstly, I'm too weak-minded to learn a big language like C++. Life just seems filled with more interesting things to understand. Secondly, I find that this specific level of manual control lets me produce better results, and faster.

The point here isn't C vs. C++ but the value of manual control for high-end professional users. It's no accident that the best cars and cameras and espresso machines in the world have manual controls. That level of on-the-spot fine-tuning often makes the difference between world-class success, and second-best.

When you are really, truly, concerned about the speed of serialization and/or the size of the result (often these contradict each other), you need hand-written binary serialization, in other words, let's hear it for Mr. Nasty!

Your basic process for writing an efficient Nasty encoder/decoder (codec) is:

- Build representative data sets and test applications that can stress-test your codec.
- Write a first dumb version of the codec.
- Test, measure, improve, and repeat until you run out of time and/or money.

Here are some of the techniques we use to make our codecs better:

- *Use a profiler.* There's simply no way to know what your code is doing until you've profiled it, for function counts and for CPU cost per function. Once you find your hot-spots, fix them.
- *Eliminate memory allocations.* On a modern Linux kernel the heap is very fast, but it's still the bottleneck in most naive codecs. On older kernels the heap can be tragically slow. Use local variables (the stack) instead of the heap where you can.
- *Test on different platforms and with different compilers and compiler options.* Apart from the heap, there are many other differences. You need to learn the main ones, and allow for these.
- *Use state to compress better.* If you are concerned about codec performance, you are almost definitely sending the same kinds of data many times. There will be redundancy between instances of data. You can detect these, and use that to compress (e.g. a short value that means "same as last time").

- *Know your data.* The best compression techniques (in terms of CPU cost for compactness) require knowing about the data. For example the techniques to compress a word list, a video, and a stream of stock market data are all different.
- *Be ready to break the rules.* Do you really need to encode integers in big-endian network byte order? x86 and ARM account for almost all modern CPUs, yet use little-endian (ARM is actually bi-endian but Android, like Windows and iOS, is little-endian).

6.7.6. Code Generation

Reading the previous two sections, you might have wondered, "could I write my own IDL generator that was better than a general-purpose one?" If this thought wandered into your mind, it probably left pretty soon after, chased by dark calculations about how much work that actually involved.

What if I told you of a way to build custom IDL generators cheaply and quickly? A way to get perfectly documented contracts, code that is as evil and domain-specific as you need, and all you need to do is sign away your soul (*who ever really used that, amirite?*) right here...

At iMatix, until a few years ago, we used code generation to build ever larger and more ambitious systems until we decided the technology (GSL) was too dangerous for common use, and we sealed the archive and locked it, with heavy chains, in a deep dungeon. Well, we actually posted it on github. If you want to try the examples that are coming up, grab the repository (https://github.com/imatix/gsl) and build yourself a `gsl` command. Typing "make" in the src subdirectory should do it (and if you're that guy who loves Windows, I'm sure you'll send a patch with project files).

This section isn't really about GSL at all, but about a useful and little-known trick that's useful for ambitious architects who want to scale themselves, as well as their work. Once you learn the trick is, you can whip up your own code generators in a short time. The code generators most software engineers know about come with a single hard-coded model. For instance, Ragel *"compiles executable finite state machines from regular languages"*, i.e. Ragel's model is a regular language. This certainly works for a good set of problems but it's far from universal. How do you describe an API in Ragel? Or a project makefile? Or even a finite-state machine like the one we used to design the Binary Star pattern in Chapter 4?

All these would benefit from code generation, but there's no universal model. So the trick is to design your own models as you need them, then make code generators as cheap compilers for that model. You need some experience in how to make good models, and you need a technology that makes it cheap to build custom code generators. Scripting languages like Perl and Python are a good option. However we actually built GSL specifically for this, and that's what I prefer.

Let's take a simple example that ties into what we already know. We'll see more extensive examples later, because I really do believe that code generation is crucial knowledge for large-scale work. In Chapter 4, we developed the Majordomo Protocol (MDP) (http://rfc.zeromq.org/spec:7), and wrote

clients, brokers, and workers for that. Now could we generate those pieces mechanically, by building our own interface description language and code generators?

When we write a GSL model, we can use *any* semantics we like, in other words we can invent domain-specific languages on the spot. I'll invent a couple - see if you can guess what they represent:

```
slideshow
    name = Cookery level 3
    page
        title = French Cuisine
        item = Overview
        item = The historical cuisine
        item = The nouvelle cuisine
        item = Why the French live longer
    page
        title = Overview
        item = Soups and salads
        item = Le plat principal
        item = BÃ©chamel and other sauces
        item = Pastries, cakes, and quiches
        item = SoufflÃ© - cheese to strawberry
```

How about this one:

```
table
    name = person
    column
        name = firstname
        type = string
    column
        name = lastname
        type = string
    column
        name = rating
        type = integer
```

The first we could compile into a presentation. The second, into SQL to create and work with a database table. So for this exercise our domain language, our model, consists of "classes" that contain "messages" that contain "fields" of various types. It's deliberately familiar. Here is the MDP client protocol:

```
<class name = "mdp_client">
    MDP/Client
    <header>
        <field name = "empty" type = "string" value = ""
            >Empty frame</field>
        <field name = "protocol" type = "string" value = "MDPC01"
            >Protocol identifier</field>
    </header>
    <message name = "request">
        Client request to broker
        <field name = "service" type = "string">Service name</field>
```

```
        <field name = "body" type = "frame">Request body</field>
    </message>
    <message name = "reply">
        Response back to client
        <field name = "service" type = "string">Service name</field>
        <field name = "body" type = "frame">Response body</field>
    </message>
</class>
```

And here is the MDP worker protocol:

```
<class name = "mdp_worker">
    MDP/Worker
    <header>
        <field name = "empty" type = "string" value = ""
            >Empty frame</field>
        <field name = "protocol" type = "string" value = "MDPW01"
            >Protocol identifier</field>
        <field name = "id" type = "octet">Message identifier</field>
    </header>
    <message name = "ready" id = "1">
        Worker tells broker it is ready
        <field name = "service" type = "string">Service name</field>
    </message>
    <message name = "request" id = "2">
        Client request to broker
        <field name = "client" type = "frame">Client address</field>
        <field name = "body" type = "frame">Request body</field>
    </message>
    <message name = "reply" id = "3">
        Worker returns reply to broker
        <field name = "client" type = "frame">Client address</field>
        <field name = "body" type = "frame">Request body</field>
    </message>
    <message name = "hearbeat" id = "4">
        Either peer tells the other it's still alive
    </message>
    <message name = "disconnect" id = "5">
        Either peer tells other the party is over
    </message>
</class>
```

GSL uses XML as its modeling language. XML has a poor reputation, having been dragged through too many enterprise sewers to smell sweet, but it has some strong positives, as long as you keep it simple. Any way to write a self-describing hierarchy of items and attributes would work.

Now here is a short IDL generator written in GSL that turns our protocol models into documentation:

```
.#  Trivial IDL generator (specs.gsl)
.#
.output "$(class.name).md"
```

```
## The $(string.trim (class.?”):left) Protocol
.for message
.   frames = count (class->header.field) + count (field)

A $(message.NAME) command consists of a multi-part message of $(frames)
frames:

.   for class->header.field
.       if name = "id"
* Frame $(item ()): 0x$(message.id:%02x) (1 byte, $(message.NAME))
.       else
* Frame $(item ()): "$(value:)" ($(string.length ("$(value)")) \
bytes, $(field.:))
.       endif
.   endfor
.   index = count (class->header.field) + 1
.   for field
* Frame $(index): $(field.?”) \
.       if type = "string"
(printable string)
.       elsif type = "frame"
(opaque binary)
.           index += 1
.       else
.           echo "E: unknown field type: $(type)"
.       endif
.       index += 1
.   endfor
.endfor
```

The XML models and this script are in the subdirectory examples/Chapter6. To do the code generation I give this command:

```
gsl -script:specs mdp_client.xml mdp_worker.xml
```

Here is the Markdown text we get for the worker protocol:

```
## The MDP/Worker Protocol

A READY command consists of a multi-part message of 4
frames:

* Frame 1: "" (0 bytes, Empty frame)
* Frame 2: "MDPW01" (6 bytes, Protocol identifier)
* Frame 3: 0x01 (1 byte, READY)
* Frame 4: Service name (printable string)

A REQUEST command consists of a multi-part message of 5
frames:

* Frame 1: "" (0 bytes, Empty frame)
* Frame 2: "MDPW01" (6 bytes, Protocol identifier)
```

```
* Frame 3: 0x02 (1 byte, REQUEST)
* Frame 4: Client address (opaque binary)
* Frame 6: Request body (opaque binary)

A REPLY command consists of a multi-part message of 5
frames:

* Frame 1: "" (0 bytes, Empty frame)
* Frame 2: "MDPW01" (6 bytes, Protocol identifier)
* Frame 3: 0x03 (1 byte, REPLY)
* Frame 4: Client address (opaque binary)
* Frame 6: Request body (opaque binary)

A HEARBEAT command consists of a multi-part message of 3
frames:

* Frame 1: "" (0 bytes, Empty frame)
* Frame 2: "MDPW01" (6 bytes, Protocol identifier)
* Frame 3: 0x04 (1 byte, HEARBEAT)

A DISCONNECT command consists of a multi-part message of 3
frames:

* Frame 1: "" (0 bytes, Empty frame)
* Frame 2: "MDPW01" (6 bytes, Protocol identifier)
* Frame 3: 0x05 (1 byte, DISCONNECT)
```

Which as you can see is close to what I wrote by hand in the original spec. Now, if you have cloned the Guide repository and you are looking at the code in examples/Chapter6, you can generate the MDP client and worker codecs. We pass the same two models to a different code generator:

```
gsl -script:codec_c mdp_client.xml mdp_worker.xml
```

Which gives us mdp_client and mdp_worker classes. Actually MDP is so simple that it's barely worth the effort of writing the code generator. The profit comes when we want to change the protocol (which we did for the standalone Majordomo project). You modify the protocol, run the command, and out pops more perfect code.

The `codec_c.gsl` code generator is not short, but the resulting codecs are much better than the hand-written code I originally put together for Majordomo. For instance the hand-written code had no error checking, and would die if you passed it bogus messages.

I'm now going to explain the pros and cons of GSL-powered model-oriented code generation. Power does not come for free and one of the greatest traps in our business is the ability to invent concepts out of thin air. GSL makes this particularly easy, so can be a particularly dangerous tool.

Do not invent concepts. The job of a designer is to remove problems, not to add features.

So, first, the advantages of model-oriented code generation:

- You can create ’perfect’ abstractions that map to your real world. So, our protocol model maps 100% to the ’real world’ of Majordomo. This would be impossible without the freedom to tune and change the model in any way.
- You can develop these perfect models quickly and cheaply.
- You can generate *any* text output. From a single model you can create documentation, code in any language, test tools, literally any output you can think of.
- You can generate (and I mean this literally) *perfect* output since it’s cheap to improve your code generators to any level you want.
- You get a single source that combines specifications and semantics.
- You can leverage a small team to a massive size. At iMatix we produced the million-line OpenAMQ messaging product out of perhaps 85K lines of input models, including the code generation scripts themselves.

Now the disadvantages:

- You add tool dependencies to your project.
- You may get carried away and create models for the pure joy of creating them.
- You may alienate newcomers to your work, who will see "strange stuff".
- You may give people a strong excuse to not invest in your project.

Cynically, model-oriented abuse works great in environments where you want to produce huge amounts of perfect code that you can maintain with little effort, and which *no-one can ever take away from you.* Personally, I like to cross my rivers and move on. But if long-term job security is your thing, this is almost perfect.

So if you do use GSL and want to create open communities around your work, here is my advice:

- Use only where you would otherwise be writing tiresome code by hand.
- Design natural models that are what people would expect to see.
- Write the code by hand first so you know what to generate.
- Do not overuse. Keep it simple! *Do not get too meta!!*
- Introduce gradually into a project.
- Put the generated code into your repositories.

We’re already using GSL in some projects around ØMQ, for example the high-level C binding, CZMQ, uses GSL to generate the socket options class (zsockopt). A 300-line code generator turns 78 lines of XML model into 1,500 lines of perfect but really boring code. That’s a good win.

6.8. Transferring Files

Let’s take a break from the lecturing and get back to our first love and the reason for doing all of this: code.

"How do I send a file?" is a common question on the ØMQ mailing lists. Not surprising, because file transfer is perhaps the oldest and most obvious type of messaging. Sending files around networks has lots of use-cases apart from annoying the copyright cartels. ØMQ is very good, out of the box, at sending events and tasks but less good at sending files.

I’ve promised, for a year or two, to write a proper explanation. Here’s a gratuitous piece of information to brighten your morning: the word "proper" comes from the archaic French "propre" which means "clean". The dark age English common folk, not being familiar with hot water and soap, changed the word to mean "foreign" or "upper-class", as in "that’s proper food!" but later the word meant just "real", as in "that’s a proper mess you’ve gotten us into!"

So, file transfer. There are several reasons you can’t just pick up a random file, blindfold it, and shove it whole into a message. The most obvious being that despite decades of determined growth in RAM sizes (and who among us old-timers doesn’t fondly remember saving up for that 1,014-byte memory extension card?!), disk sizes obstinately remain much larger. Even if we could send a file with one instruction (say, using a system call like sendfile), we’d hit the reality that networks are not infinitely fast, nor perfectly reliable. After trying to upload a large file several times on a slow flaky network (WiFi, anyone?), you’ll realize that a proper file transfer protocol needs a way to recover from failures. That is, a way to send only the part of a file that wasn’t yet received.

Finally, after all this, if you build a proper file server, you’ll notice that simply sending massive amounts of data to lots of clients creates that situation we like to call, in the technical parlance, "*server went belly-up due to all available heap memory being eaten by a poorly-designed application*". A proper file transfer protocol needs to pay attention to memory use.

We’ll solve these problems properly, one by one, which should hopefully get us to a good and proper file transfer protocol running over ØMQ. First, let’s generate a 1GB test file with random data (real power-of-two-giga-like-Von-Neumman-intended, not the fake silicon ones the memory industry likes to sell):

```
dd if=/dev/urandom of=testdata bs=1M count=1024
```

This is large enough to be troublesome when we have lots of clients asking for the same file at once, and on many machines, 1GB is going to be too large to allocate in memory anyhow. As a base reference, let’s measure how long it takes to copy this file from disk back to disk. This will tell us how much our file transfer protocol adds on top (including ’network’ costs):

```
$ time cp testdata testdata2

real    0m7.143s
```

```
user    0m0.012s
sys     0m1.188s
```

The 4-figure precision is misleading; expect variations of 25% either way. This is just an "order of magnitude" measurement.

Here's our first cut at the code, where the client asks for the test data and the server just sends it, without stopping for breath, as a series of messages, where each message holds one 'chunk':

Example 6-1. File transfer test, model 1 (fileio1.lua)

```
(This example still needs translation into Lua)
```

It's pretty simple but we already run into a problem: if we send too much data to the ROUTER socket, we can easily overflow it. The simple but stupid solution is to put an infinite high-water mark on the socket. It's stupid because we now have no protection against exhausting the server's memory. Yet without an infinite HWM we risk losing chunks of large files.

Try this: set the HWM to 1,000 (in ØMQ/3.x this is the default) and then reduce the chunk size to 100K so we send 10K chunks in one go. Run the test, and you'll see it never finishes. As the zmq_socket[3] man page says with cheerful brutality, for the ROUTER socket: "*ZMQ_HWM option action: Drop*".

We have to control the amount of data the server sends up-front. There's no point in it sending more than the network can handle. Let's try sending one chunk at a time. In this version of the protocol, the client will explicitly say,"give me chunk N", and the server will fetch that specific chunk from disk and send it.

Here's the improved second model, where the client asks for one chunk at a time, and the server only sends one chunk for each request it gets from the client:

Example 6-2. File transfer test, model 2 (fileio2.lua)

```
(This example still needs translation into Lua)
```

It is much slower now, because of the to-and-fro chatting between client and server. We pay about 300 microseconds for each request-reply round-trips, on a local loop connection (client and server on the same box). It doesn't sound like much but it adds up quickly:

```
$ time ./fileio1
4296 chunks received, 1073741824 bytes

real    0m0.669s
user    0m0.056s
sys     0m1.048s

$ time ./fileio2
4295 chunks received, 1073741824 bytes
```

```
real    0m2.389s
user    0m0.312s
sys     0m2.136s
```

There are two valuable lessons here. First, while request-reply is easy, it's also too slow for high-volume data flows. Paying that 300 microseconds once would be fine. Paying it for every single chunk isn't acceptable, particularly on real networks with latencies of perhaps 1,000 times higher.

The second point is something I've said before but will repeat: it's incredibly easy to experiment, measure, and improve our protocols over ØMQ. And when the cost of something comes way down, you can afford a lot more of it. Do learn to develop and prove your protocols in isolation: I've seen teams waste time trying to improve poorly-designed protocols that are too deeply embedded in applications to be easily testable or fixable.

Our model 2 file transfer protocol isn't so bad, apart from performance:

- It completely eliminates any risk of memory exhaustion. To prove that we set the high-water mark to 1 in both sender and receiver.
- It lets the client choose the chunk size, which is useful because if there's any tuning of the chunk size to be done, for network conditions, for file types, or to reduce memory consumption further, it's the client that should be doing this.
- It gives us fully restartable file transfers.
- It allows the client to cancel the file transfer at any point in time.

If we just didn't have to do a request for each chunk, it'd be a usable protocol. What we need is a way for the server to send multiple chunks, without waiting for the client to request or acknowledge each one. What are the options?

- The server could send 10 chunks at once, then wait for a single acknowledgment. That's exactly like multiplying the chunk size by 10, so pointless. And yes, it's just as pointless for all values of 10.
- The server could send chunks without any chatter from the client but with a slight delay between each send, so that it would send chunks only as fast as the network could handle them. This would require the server to know what's happening at the network layer, which sounds like hard work. It also breaks layering horribly. And what happens if the network is really fast but the client itself is slow? Where are chunks queued then?
- The server could try to spy on the sending queue, i.e. see how full it is, and send only when the queue isn't full. Well, ØMQ doesn't allow that because it doesn't work, for the same reason as throttling doesn't work. The server and network may be more than fast enough, but the client may be a slow little device.
- We could modify libzmq to take some other action on reaching HWM. Perhaps it could block? That would mean that a single slow client would block the whole server, so no thank you. Maybe it could return an error to the caller? Then the server could do something smart like... well, there isn't really anything it could do that's any better than dropping the message.

Apart from being complex and variously unpleasant, none of these options would even work. What we need is a way for the client to tell the server, asynchronously and in the background, that it's ready for more. Some kind of asynchronous flow control. If we do this right, data should flow without interruption from the server to the client, but only as long as the client is reading it. Let's review our three protocols. This was the first one:

```
C: fetch
S: chunk 1
S: chunk 2
S: chunk 3
....
```

And the second introduced a request for each chunk:

```
C: fetch chunk 1
S: send chunk 1
C: fetch chunk 2
S: send chunk 2
C: fetch chunk 3
S: send chunk 3
C: fetch chunk 4
....
```

Now - waves hands mysteriously - here's a changed protocol that fixes the performance problem:

```
C: fetch chunk 1
C: fetch chunk 2
C: fetch chunk 3
S: send chunk 1
C: fetch chunk 4
S: send chunk 2
S: send chunk 3
....
```

It looks suspiciously similar. In fact it's identical except that we send multiple requests without waiting for a reply for each one. This is a technique called "pipelining" and it works because our DEALER and ROUTER sockets are fully asynchronous.

Here's the third model of our file transfer test-bench, with pipelining. The client sends a number of requests ahead (the "credit") and then each time it processes an incoming chunk, it sends one more credit. The server will never send more chunks than the client has asked for:

Example 6-3. File transfer test, model 3 (fileio3.lua)

```
(This example still needs translation into Lua)
```

What we've achieved here, with a little magic, is to take control of the end-to-end pipeline including all network buffers and ØMQ queues at sender and receiver, and then ensure that pipeline is always filled

with data while never growing beyond a predefined limit. More than that, the client decides exactly when to send "credit" to the sender. It could be when it receives a chunk, or when it has fully processed a chunk. And this happens asynchronously, with no significant performance cost.

In the third model I chose a pipeline size of 10 messages (each message is a chunk). This will cost a maximum of 2.5MB memory per client. So with 1GB of memory we can handle at least 400 clients. We can try to calculate the ideal pipeline size. It takes about 0.7 seconds to send the 1GB file, which is about 160 microseconds for a chunk. A round trip is 300 microseconds, so the pipeline needs to be at least 3-5 to keep the server busy. In practice, I still got performance spikes with a pipeline of 5, probably because the credit messages sometimes get delayed by outgoing data. So at 10, it works consistently.

```
$ time ./fileio3
4291 chunks received, 1072741824 bytes

real    0m0.777s
user    0m0.096s
sys     0m1.120s
```

Do measure rigorously. Your calculations may be good but the real world tends to have its own opinions.

What we've made is clearly not yet a real file transfer protocol, but it proves the pattern and I think it is the simplest plausible design. For a real working protocol we'd want to add some or all of:

- Authentication and access controls, even without encryption: the point isn't to protect sensitive data but to catch errors like sending test data to production servers.
- A Cheap-style request including file path, optional compression, and other stuff we've learned is useful from HTTP (such as If-Modified-Since).
- A Cheap-style response, at least for the first chunk, that provides meta data such as file size (so the client can pre-allocate and avoid unpleasant disk-full situations).
- The ability to fetch a set of files in one go, otherwise the protocol becomes inefficient for large sets of small files.
- Confirmation from the client when it's fully received a file, to recover from chunks that might be lost of the client disconnects unexpectedly.

So far, our semantic has been "fetch"; that is, the recipient knows (somehow), that they need a specific file, so they ask for it. The knowledge of which files exist, and where they are is then passed out-of-band (e.g. in HTTP, by links in the HTML page).

How about a "push" semantic? There are two plausible use-cases for this. First, if we adopt a centralized architecture with files on a main "server" (not something I'm advocating, but people do sometimes like this), then it's very useful to allow clients to upload files to the server. Second, it lets do a kind of pub-sub for files, where the client asks for all new files of some type; as the server gets these, it forwards them to the client.

A fetch semantic is synchronous, while a push semantic is asynchronous. Asynchronous is less chatty, so faster. Also, you can do cute things like "*subscribe to this path*" so creating a publish-subscribe file transfer architecture. That is so obviously awesome that I shouldn't need to explain what problem it solves.

Still, here is the problem with the fetch semantic: that out-of-band route to tell clients what files exist. No matter how you do this, it ends up complex. Either clients have to poll, or you need a separate pub-sub channel to keep clients up to date, or you need user interaction.

Thinking this through a little more, though, we can see that fetch is just a special case of publish-subscribe. So we can get the best of both worlds. Here is the general design:

- Fetch this path
- Here is credit (repeat)

To make this work (and we will, my dear readers), we need to be a little more explicit about how we send credit to the server. The cute trick of treating a pipelined "fetch chunk" request as credit won't fly since the client doesn't know any longer what files actually exist, how large they are, anything. If the client says, "I'm good for 250,000 bytes of data", this should work equally for one file of 250K bytes, or 100 files of 2,500 bytes.

And this gives us "credit-based flow control", which effectively removes the need for HWMs, and any risk of memory overflow.

6.9. Heartbeating

Just as a real protocol needs to solve the problem of flow control, it also needs to solve the problem of knowing whether a peer is alive or dead. This is not an issue specific to ØMQ. TCP has a long timeout (30 minutes or so), that means that it can be impossible to know whether a peer has died, been disconnected, or gone on a weekend to Prague with a case of vodka, a redhead, and a large expense account.

Heartbeating is not easy to get right, and as with flow control it can make the difference between a working, and failing architecture. So using our standard approach, let's start with the simplest possible heartbeat design, and develop better and better designs until we have one with no visible faults.

6.9.1. Shrugging It Off

A decent first iteration is to do no heartbeating at all and see what actually happens. Many if not most ØMQ applications do this. ØMQ encourages this by hiding peers in many cases. What problems does this approach cause?

- When we use a ROUTER socket in an application that tracks peers, as peers disconnect and reconnect, the application will leak memory and get slower and slower.
- When we use SUB or DEALER-based data recipients, we can't tell the difference between good silence (there's no data) and bad silence (the other end died). When a recipient knows the other side died, it can for example switch over to a backup route.
- If we use a TCP connection that stays silent for a long while, it will, in some networks, just die. Sending something (technically, a "keep-alive" more than a heartbeat), will keep the network alive.

6.9.2. One-Way Heartbeats

So, our first solution is to sending a "heartbeat" message from each node to its peers, every second or so. When one node hears nothing from another, within some timeout (several seconds, typically), it will treat that peer as dead. Sounds good, right? Sadly no. This works in some cases but has nasty edge cases in other cases.

For PUB-SUB, this does work, and it's the only model you can use. SUB sockets cannot talk back to PUB sockets, but PUB sockets can happily send "I'm alive" messages to their subscribers.

As an optimization, you can send heartbeats only when there is no real data to send. Furthermore, you can send heartbeats progressively slower and slower, if network activity is an issue (e.g. on mobile networks where activity drains the battery). As long as the recipient can detect a failure (sharp stop in activity), that's fine.

Now the typical problems with this design:

- It can be inaccurate when we send large amounts of data, since heartbeats will be delayed behind that data. If heartbeats are delayed, you can get false timeouts and disconnections due to network congestion. Thus, always treat *any* incoming data as a heartbeat, whether or not the sender optimizes out heartbeats.
- While the PUB-SUB pattern will drop messages for disappeared recipients, PUSH and DEALER sockets will queue them. So, if you send heartbeats to a dead peer, and it comes back, it'll get all the heartbeats you sent. Which can be thousands. Whoa, whoa!
- This design assumes that heartbeat timeouts are the same across the whole network. But that won't be accurate. Some peers will want very aggressive heart-beating, to detect faults rapidly. And some will want very relaxed heart-beating, to let sleeping networks lie, and save power.

6.9.3. Ping-Pong Heartbeats

Our third design uses a ping-pong dialog. One peer sends a ping command to the other, which replies with a pong command. Neither command has any payload. Pings and pongs are not correlated. Since the roles of "client" and "server" are often arbitrary, we specify that either peer can in fact send a ping and

expect a pong in response. However, since the timeouts depend on network topologies known best to dynamic clients, it is usually the client which pings the server.

This works for all ROUTER-based brokers. The same optimizations we used in the second model make this work even better: treat any incoming data as a pong, and only send a ping when not otherwise sending data.

6.10. State Machines

Software engineers tend to treat (finite) state machines as a kind of intermediary interpreter. That is, you take a regular language and compile that into a state machine, then execute the state machine. The state machine itself is rarely visible to the developer: it's an internal representation, optimized, compressed, and bizarre.

However it turns out that state machines are also valuable as a first-class modeling languages for protocol handlers, i.e. ØMQ clients and servers. ØMQ makes it rather easy to design protocols, but we've never defined a good pattern for writing those clients and servers properly.

A protocol has at least two levels:

- How we represent individual messages on the wire.
- How messages flow between peers, and the significance of each message.

We've seen in this chapter how to produce codecs that handle serialization. That's a good start. But if we leave the second job to developers, that gives them a lot of room to interpret. As we make more ambitious protocols (file transfer + heart-beating + credit + authentication), it becomes less and less sane to try to implement clients and servers by hand.

Yes, people do this almost systematically. But the costs are high, and they're avoidable. I'll explain how to model protocols using state machines, and how to generate neat and solid code from those models.

My experience with using state machines as a software construction tool dates to 1985 and my first real job making tools for application developers. In 1991 I turned that knowledge into a free software tool called Libero, which spat out executable state machines from a simple text model.

The thing about Libero's model was that it was readable. That is, you described your program logic as named states, each accepting a set of events, each doing some real work. The resulting state machine hooked into your application code, driving it like a boss.

Libero was charmingly good at its job, fluent in many languages, and modestly popular given the enigmatic nature of state machines. We used Libero in anger in dozens of large distributed applications,

one of which was finally switched off in 2011. State-machine driven code construction worked so well that it's somewhat impressive this approach never hit the mainstream of software engineering.

So in this section I'm going to explain Libero's model, and show how to use it to generate ØMQ clients and servers. We'll use GSL again but like I said, the principles are general and you can put together code generators using any scripting language.

As a worked example let's see how to carry-on a stateful dialog with a peer on a ROUTER socket. We'll develop the server using a state machine (and the client by hand). We have a simple protocol that I'll call "NOM". I'm using the oh-so-very-serious keywords for unprotocols (http://unprotocols.org/blog:2) proposal:

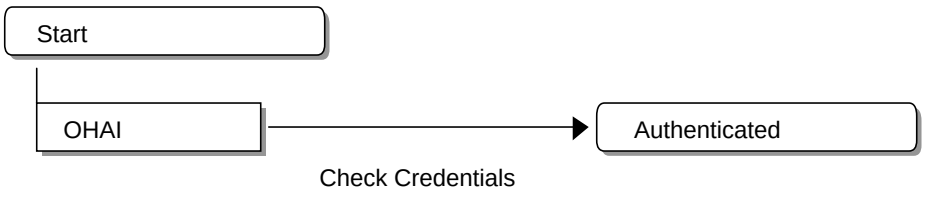
```
nom-protocol    = open-peering *use-peering

open-peering    = C:OHAI ( S:OHAI-OK / S:WTF )

use-peering     = C:ICANHAZ
                / S:CHEEZBURGER
                / C:HUGZ S:HUGZ-OK
                / S:HUGZ C:HUGZ-OK
```

I've not found a quick way to explain the true nature of state machine programming. In my experience, it invariably takes a few days of practice. After three or four days' exposure to the idea there is a near-audible 'click!' as something in the brain connects all the pieces together. We'll make it concrete by looking at the state machine for our NOM server.

A useful thing about state machines is that you can read them state by state. Each state has a unique descriptive name, and one or more *events*, which we list in any order. For each event we perform zero or more *actions*, and we then move to a *next state* (or stay in the same state).

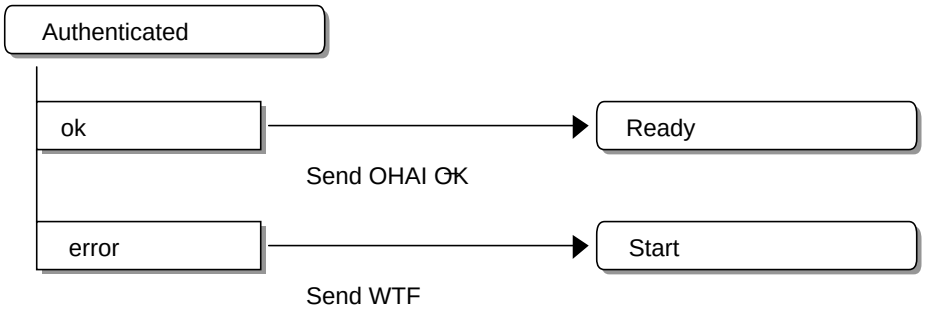
In a ØMQ protocol server, we have a state machine instance *per client*. That sounds complex but it isn't, as we'll see. We describe our first state (Start) as having one valid event, "OHAI". We check the user's credentials and then arrive in the Authenticated state(Figure 6-1).

Figure 6-1. The 'Start' State



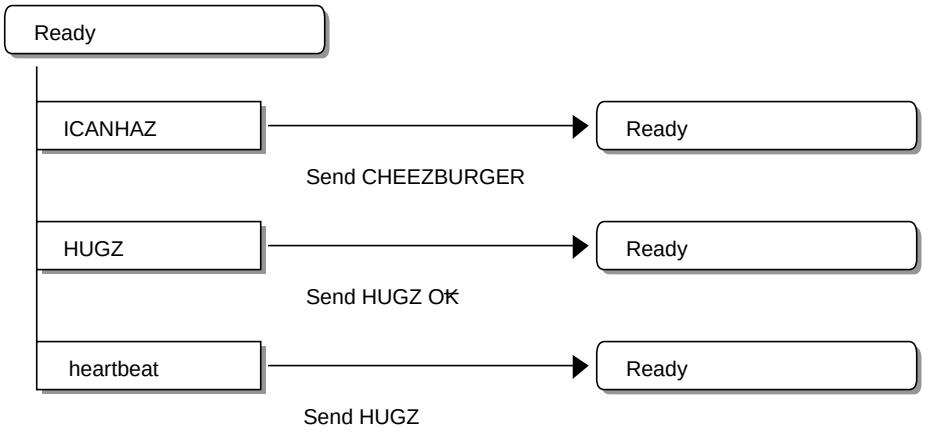
The Check Credentials action produces either an ’ok’ or an ’error’ event. It’s in the Authenticated state that we handle these two possible events, by sending an appropriate reply back to the client(Figure 6-2). If authentication failed, we return to the Start state where the client can try again.

Figure 6-2. The ’Authenticated’ State



When authentication has succeeded, we arrive in the Ready state. Here we have three possible events: an ICANHAZ or HUGZ message from the client, or a heartbeat timer event(Figure 6-3).

Figure 6-3. The ’Ready’ State



There are a few more things about this state machine model that are worth knowing:

- Events in upper case (like "HUGZ") are ’external events’ that come from the client as messages.
- Events in lower case (like "heartbeat") are ’internal events’, produced by code in the server.
- The "Send SOMETHING" actions are shorthand for sending a specific reply back to the client.

- Events that aren't defined in a particular state are silently ignored.

Now, the original source for these pretty pictures is an XML model:

```
<class name = "nom_server" script = "server_c">

<state name = "start">
    <event name = "OHAI" next = "authenticated">
        <action name = "check credentials" />
    </event>
</state>

<state name = "authenticated">
    <event name = "ok" next = "ready">
        <action name = "send" message ="OHAI-OK" />
    </event>
    <event name = "error" next = "start">
        <action name = "send" message = "WTF" />
    </event>
</state>

<state name = "ready">
    <event name = "ICANHAZ">
        <action name = "send" message = "CHEEZBURGER" />
    </event>
    <event name = "HUGZ">
        <action name = "send" message = "HUGZ-OK" />
    </event>
    <event name = "heartbeat">
        <action name = "send" message = "HUGZ" />
    </event>
</state>
</class>
```

The code generator is in examples/Chapter6/server_c.gsl. It is a fairly complete tool that I'll use and expand for more serious work later. It generates:

- A server class in C (nom_server.c, nom_server.h) that implements the whole protocol flow.
- A selftest method that runs the selftest steps listed in the XML file.
- Documentation in the form of graphics (the pretty pictures).

Here's a simple C main program that starts the generated NOM server:

```
#include "czmq.h"
#include "nom_server.h"

int main (int argc, char *argv [])
{
    printf ("Starting NOM protocol server on port 6000...\n");
    nom_server_t *server = nom_server_new ();
```

```
    nom_server_bind (server, "tcp://*:6000");
    nom_server_wait (server);
    nom_server_destroy (&server);
    return 0;
}
```

The generated nom_server class is a fairly classic model. It accepts client messages on a ROUTER socket. The first frame on every request is the client's identity. The server manages a set of clients, each with state. As messages arrive, it feeds these as 'events' to the state machine. Here's the core of the state machine, as a mix of GSL commands and the C code we intend to generate:

```
client_execute (client_t *self, int event)
{
    self->next_event = event;
    while (self->next_event) {
        self->event = self->next_event;
        self->next_event = 0;
        switch (self->state) {
.for class.state
            case $(name:c)_state:
.   for event
.       if index () > 1
                else
.       endif
                if (self->event == $(name:c)_event) {
.       for action
.           if name = "send"
                    zmsg_addstr (self->reply, "$(message:)");
.           else
                $(name:c)_action (self);
.           endif
.       endfor
.       if defined (event.next)
                    self->state = $(next:c)_state;
.       endif
                }
.   endfor
                break;
.endfor
        }
        if (zmsg_size (self->reply) > 1) {
            zmsg_send (&self->reply, self->router);
            self->reply = zmsg_new ();
            zmsg_add (self->reply, zframe_dup (self->address));
        }
    }
}
```

Each client is held as an object with various properties, including the variables we need to represent a state machine instance:

```
event_t next_event;         //  Next event
state_t state;              //  Current state
event_t event;              //  Current event
```

You will see by now that we are generating technically-perfect code that has the precise design and shape we want. The only clue that the nom_server class isn't hand-written is that the code is *too good*. People who complain that code generators produce poor code are obviously used to poor code generators. It is trivial to extend our model as we need it. For example, here's how we generate the selftest code.

First, we add a "selftest" item to the state machine and write our tests. We're not using any XML grammar or validators so it really is just a matter of opening the editor and adding half-a-dozen lines of text:

```
<selftest>
    <step send = "OHAI" body = "Sleepy" recv = "WTF" />
    <step send = "OHAI" body = "Joe" recv = "OHAI-OK" />
    <step send = "ICANHAZ" recv = "CHEEZBURGER" />
    <step send = "HUGZ" recv = "HUGZ-OK" />
    <step recv = "HUGZ" />
</selftest>
```

Designing on the fly, I decided that "send" and "recv" were a nice way to express "send this request, then expect this reply". Here's the GSL code that turns this model into real code:

```
.for class->selftest.step
.   if defined (send)
    msg = zmsg_new ();
    zmsg_addstr (msg, "$(send:)");
.       if defined (body)
    zmsg_addstr (msg, "$(body:)");
.       endif
    zmsg_send (&msg, dealer);

.   endif
.   if defined (recv)
    msg = zmsg_recv (dealer);
    assert (msg);
    command = zmsg_popstr (msg);
    assert (streq (command, "$(recv:)"));
    free (command);
    zmsg_destroy (&msg);

.   endif
.endfor
```

Finally, one of the more tricky but absolutely essential parts of any state machine generator is *how do I plug this into my own code?* As a minimal example for this exercise I wanted to implement the "check credentials" action by accepting all OHAIs from my friend Joe (Hi Joe!) and reject everyone else's

OHAIs. After some thought I decided to grab code directly from the state machine model. So in nom_server.xml, you'll see this:

```
<action name = "check credentials">
    char *body = zmsg_popstr (self->request);
    if (body && streq (body, "Joe"))
        self->next_event = ok_event;
    else
        self->next_event = error_event;
    free (body);
</action>
```

And the code generator grabs that custom code and inserts it into the generated nom_server.c file:

```
.for class.action
static void
$(name:c)_action (client_t *self) {
$(string.trim (.):)
}
.endfor
```

And now we have something quite elegant: a single source file that describes my server state machine, and which also contains the native implementations for my actions. A nice mix of high-level and low-level that is about 90% smaller than the C code.

Beware, as your head spins with notions of all the amazing things you could produce with such leverage. While this approach gives you real power, it also moves you away from your peers, and if you go too far, you'll find yourself working alone.

By the way, this simple little state machine design exposes just three variables to our custom code:

- `self->next_event`
- `self->request`
- `self->reply`

In the Libero state machine model there are a few more concepts that we've not used here, but which we will need when we write larger state machines:

- Exceptions, which lets us write terser state machines. When an action raises an exception, further processing on the event stops. The state machine can then define how to handle exception events.
- Defaults state, where we can define default handling for events (especially useful for exception events).

6.11. Authentication using SASL

When we designed AMQP in 2007, we chose SASL (http://en.wikipedia.org/wiki/Simple_Authentication_and_Security_Layer) for the authentication layer, one of the ideas we took from the BEEP protocol framework. SASL looks complex at first, but it's simple and fits very nicely into a ØMQ-based protocol. What I especially like about SASL is that it's scalable. You can start with anonymous access, or plain text authentication and no security, and grow to more secure mechanisms over time, without changing your protocol one bit.

I'm not going to give a deep explanation now, since we'll see SASL in action somewhat later. But I'll explain the principle so you're already somewhat prepared.

In the NOM protocol the client started with an OHAI command, which the server either accepted ("Hi Joe!") or rejected. This is simple but not scalable since server and client have to agree upfront what kind of authentication they're going to do.

What SASL introduced, and which is genius, is a fully abstracted and negotiable security layer that's still easy to implement at the protocol level. It works as follows:

- The client connects.
- The server challenges the client, passing a list of security "mechanisms" that it knows about.
- The client chooses a security mechanism that it knows about, and answers the server's challenge with a blob of opaque data that (and here's the neat trick) some generic security library calculates and gives to the client.
- The server takes the security mechanism the client choose, and that blob of data, and passes it to its own security library.
- The library either accepts the client's answer, or the server challenges again.

There are a number of free SASL libraries. When we come to real code, we'll implement just two mechanisms, ANONYMOUS and PLAIN, which don't need any special libraries.

To support SASL we have to add an optional challenge/response step to our "open-peering" flow. Here is what the resulting protocol grammar looks like (I'm modifying NOM to do this):

```
secure-nom      = open-peering *use-peering

open-peering    = C:OHAI *( S:ORLY C:YARLY ) ( S:OHAI-OK / S:WTF )

ORLY            = 1*mechanism challenge
mechanism       = string
challenge       = *OCTET

YARLY           = mechanism response
response        = *OCTET
```

Where ORLY and YARLY contain a string (a list of mechanisms in ORLY, one mechanism in YARLY) and a blob of opaque data. Depending on the mechanism, the initial challenge from the server may be empty. We don't care a jot: we just pass this to the security library to deal with.

The SASL RFC (http://tools.ietf.org/html/rfc4422) goes into detail about other features (that we don't need), the kinds of ways SASL could be attacked, and so on.

Unless you're a security geek, all you should care about is the impact on the protocol, which is as simple as I've explained here.