Chapter One

Chapter 1 - Basics

top prev next

Fixing the World

top prev next

How to explain ZeroMQ? 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 ZeroMQ at all, because that's most likely where you, the reader, still are today.

Programming is 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" 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 and our knowledge is disappearing from our personal computers into clouds that we cannot access and with which we cannot compete. Who owns our social networks? It is like the mainframe-PC revolution in reverse.

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

There have been many attempts to solve the challenge of connected code. 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, and 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.

Starting Assumptions

top prev next

We assume you are using at least version 3.2 of ZeroMQ. We assume you are using a Linux box or something similar. We assume you can read C code, more or less, as that's the default language for the examples. We assume that when we write constants like PUSH or SUBSCRIBE, you can imagine they are really called `ZMQ_PUSH` or `ZMQ_SUBSCRIBE` if the programming language needs it.

Getting the Examples

top prev next

The examples live in a public GitHub repository. The simplest way to get all the examples is to clone this repository:

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

Next, browse the examples subdirectory. You'll find examples by language. If there are examples missing in a language you use, you're encouraged to submit a translation. This is how this text became so useful, thanks to the work of many people. All examples are licensed under MIT/X11.

Ask and Ye Shall Receive

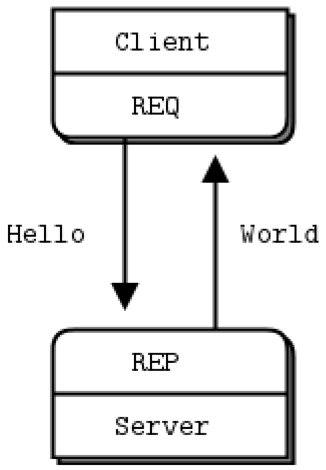
top prev next

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

hwserver: Hello World server in PHP

C | C++ | C# | Clojure | CL | Delphi | Erlang | F# | Felix | Go | Haskell | Haxe | Lua | Node.js | Objective-C | Perl | Python | Q | Racket | Ruby | Scala | Tcl | Ada | Basic | Java | ooc

Figure 2 - Request-Reply



The REQ-REP socket pair is in lockstep. The client issues `zmq_send()` and then `zmq_recv()`, 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 issues `zmq_recv()` and then `zmq_send()` in that order, as often as it needs to.

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

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

```
#include <string>
#include <iostream>
#ifndef _WIN32
#include <unistd.h>
#else
#include <windows.h>

#define sleep(n)        Sleep(n)
#endif

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 (reply.data (), "World", 5);
                socket.send (reply);
        }
        return 0;
}
```

hwserver.cpp: Hello World server

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

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

$context = new ZMQContext(1);
```

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

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

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

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

hwserver.php: Hello World server

```c
//  Hello World server

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

int main (void)
{
    //  Socket to talk to clients
    void *context = zmq_ctx_new ();
    void *responder = zmq_socket (context, ZMQ_REP);
    int rc = zmq_bind (responder, "tcp://*:5555");
    assert (rc == 0);

    while (1) {
        char buffer [10];
        zmq_recv (responder, buffer, 10, 0);
        printf ("Received Hello\n");
        sleep (1);          //  Do some 'work'
        zmq_send (responder, "World", 5, 0);
    }
    return 0;
}
```

hwserver.c: Hello World server

The server in other languages:

hwserver: Hello World server in PHP

C | C++ | C# | Clojure | CL | Delphi | Erlang | F# | Felix | Go | Haskell | Haxe | Lua | Node.js | Objective-C | Perl | Python | Q | Racket | Ruby | Scala | Tcl | Ada | Basic | Java | ooc

Here's the client code:

hwclient: Hello World client in PHP

C | C++ | C# | Clojure | CL | Delphi | Erlang | F# | Felix | Go | Haskell | Haxe | Lua | Node.js | Objective-C | Perl | Python | Q | Racket | Ruby | Scala | Tcl | Ada | Basic | Java | ooc

Now this looks too simple to be realistic, but ZeroMQ sockets have, as we already learned, superpowers. You could 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 us explain briefly what these two programs are actually doing. They create a ZeroMQ 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 we won't cover it until Chapter 4 - Reliable Request-Reply Patterns.

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 a heavy load. This is the request-reply pattern, probably the simplest way to use ZeroMQ. It maps to RPC and the classic client/server model.

A Minor Note on Strings

top prev next

ZeroMQ 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:

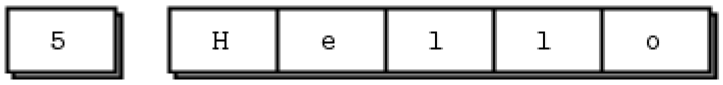
```
zmq_send (requester, "Hello", 6, 0);
```

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 3 - A ZeroMQ 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. When your client and server don't agree on the string format, you will get weird results.

When you receive string data from ZeroMQ 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 **ZeroMQ 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 ZeroMQ string maps neatly to a ZeroMQ 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 ZeroMQ string and deliver it to the application as a valid C string:

```
//  Receive ZeroMQ string from socket and convert into C string
//  Chops string at 255 chars, if it's longer
static char *
s_recv (void *socket) {
    char buffer [256];
    int size = zmq_recv (socket, buffer, 255, 0);
    if (size == -1)
        return NULL;
    if (size > 255)
        size = 255;
    buffer [size] = 0;
    return strdup (buffer);
}
```

This makes a 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 ZeroMQ format, and package this into a header file we can reuse.

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

Version Reporting

top prev next

ZeroMQ 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 ZeroMQ you're actually linking with.

Here is a tiny program that does that:

version: ZeroMQ version reporting in PHP

C | C++ | C# | Clojure | CL | Delphi | Erlang | F# | Felix | Go | Haskell | Java | Lua | Node.js | Objective-C | Perl | Python | Q | Ruby | Scala | Tcl | Ada | Basic | Haxe | ooc | Racket

Getting the Message Out

top prev next

The second classic pattern is one-way data distribution, in which a server pushes updates to a set of clients. Let's see an example that pushes out weather updates consisting of a 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:

wuserver: Weather update server in PHP

C | C++ | C# | Clojure | CL | Delphi | Erlang | F# | Felix | Go | Haskell | Haxe | Java | Lua | Node.js | Objective-C | Perl | Python | Racket | Ruby | Scala | Tcl | Ada | Basic | ooc | Q

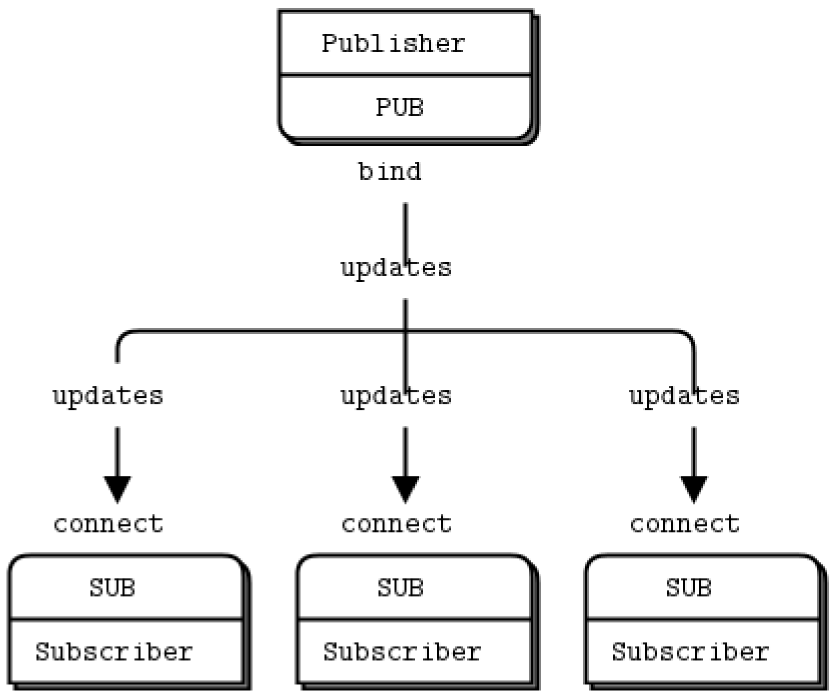
There's no start and no end to this stream of updates, it's like a never ending broadcast.

Here is the 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:

wuclient: Weather update client in PHP

C | C++ | C# | Clojure | CL | Delphi | Erlang | F# | Felix | Go | Haskell | Haxe | Java | Lua | Node.js | Objective-C | Perl | Python | Racket | Ruby | Scala | Tcl | Ada | Basic | ooc | Q

Figure 4 - Publish-Subscribe



Note that when you use a SUB socket you **must** set a subscription using `zmq_setsockopt()` 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 an update matches ANY subscription, the subscriber receives it. The subscriber can also cancel specific subscriptions. A subscription is often, but not necessarily a printable string. See `zmq_setsockopt()` for how this works.

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

In theory with ZeroMQ 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 we're going to explain it in detail. Remember that ZeroMQ 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 from handshaking that takes several milliseconds depending on your network and the number of hops between peers. In that time, ZeroMQ can send 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 2 - Sockets and Patterns we'll explain how to synchronize a publisher and subscribers so that you don't start to publish data until the subscribers really are connected and ready. There is a simple and stupid way to delay the publisher, which is to sleep. Don't do this in a real application, though, because 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 - Sockets and Patterns 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. One also assumes that the subscriber doesn't care what transpired before it started up. This is how we built our weather client example.

So the client subscribes to its chosen zip code and collects 100 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 hundred updates, it calculates the average, prints it, and exits.

Some points about the publish-subscribe (pub-sub) 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.
- From ZeroMQ v3.x, filtering happens at the publisher side when using a connected protocol (`tcp://` or `ipc://`). Using the `epgm://` protocol, filtering happens at the subscriber side. In ZeroMQ v2.x, all filtering happened at the subscriber side.

This is how long it takes to receive and filter 10M messages on my laptop, which is an 2011-era Intel i5, decent but nothing special:

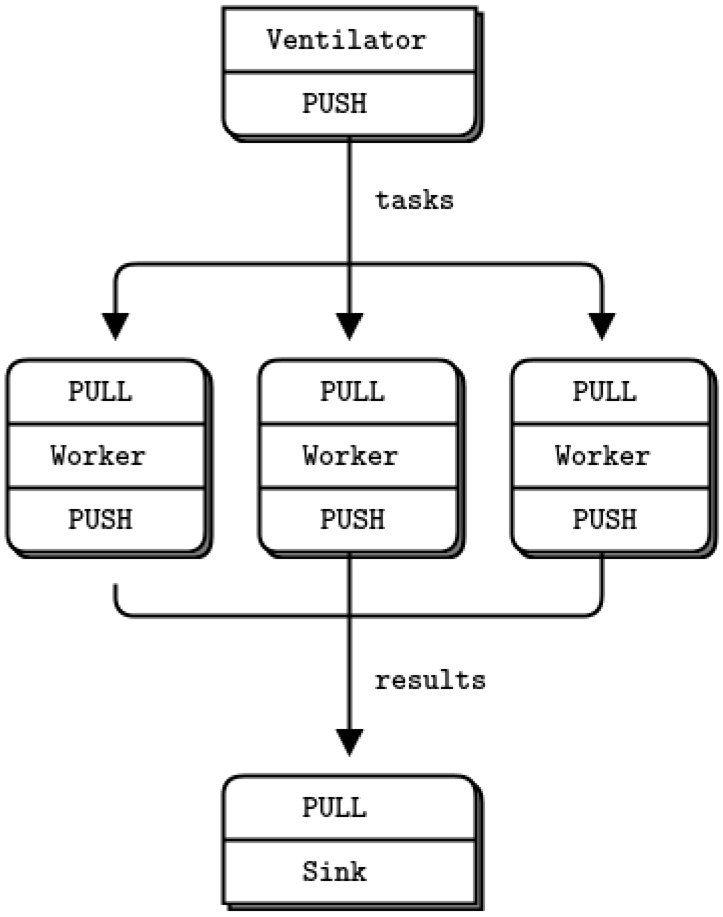
```
$ time wuclient
Collecting updates from weather server...
Average temperature for zipcode '10001 ' was 28F

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

Divide and Conquer

top prev next

Figure 5 - Parallel Pipeline



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

- A ventilator that produces tasks that can be done in parallel
- A set of workers that process tasks
- 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 math. Here is the ventilator. It generates 100 tasks, each a message telling the worker to sleep for some number of milliseconds:

taskvent: Parallel task ventilator in PHP

C | C++ | C# | Clojure | CL | Delphi | Erlang | F# | Felix | Go | Haskell | Haxe | Java | Lua | Node.js | Objective-C | Perl | Python | Ruby | Scala | Tcl | Ada | Basic | ooc | Q | Racket

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

taskwork: Parallel task worker in PHP

C | C++ | C# | Clojure | CL | Delphi | Erlang | F# | Felix | Go | Haskell | Haxe | Java | Lua | Node.js | Objective-C | Perl | Python | Ruby | Scala | Tcl | Ada | Basic | ooc | Q | Racket

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:

tasksink: Parallel task sink in PHP

C | C++ | C# | Clojure | CL | Delphi | Erlang | F# | Felix | Go | Haskell | Haxe | Java | Lua | Node.js | Objective-C | Perl | Python | Ruby | Scala | Tcl | Ada | Basic | ooc | Q | Racket

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

- 1 worker: total elapsed time: 5034 msecs.
- 2 workers: total elapsed time: 2421 msecs.
- 4 workers: total elapsed time: 1018 msecs.

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

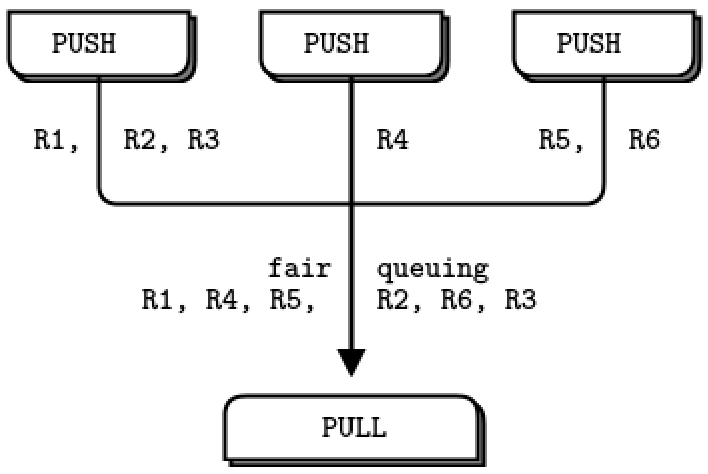
- The workers connect upstream to the ventilator, and downstream to the sink. This means you can add workers arbitrarily. If the workers bound to their endpoints, you would need (a) more endpoints and (b) to modify the ventilator and/or the sink each time you added a worker. We say that the ventilator 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 ZeroMQ and there is no easy solution. The `zmq_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 in the ventilator, and see what happens.

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

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

Figure 6 - Fair Queuing



The pipeline pattern also exhibits the "slow joiner" syndrome, leading to accusations that PUSH sockets don't load balance properly. If you are using PUSH and PULL, and one of your workers gets way more messages than the others, it's because that PULL socket has joined faster than the others, and grabs a lot of messages before the others manage to connect. If you want proper load balancing, you probably want to look at the load balancing pattern in Chapter 3 - Advanced Request-Reply Patterns.

Programming with ZeroMQ

top prev next

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

- Learn ZeroMQ step-by-step. It's just one simple API, but it hides a world of possibilities. Take the possibilities slowly and 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 and 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 ZeroMQ 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. ZeroMQ 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.

Getting the Context Right

top prev next

ZeroMQ applications always start by creating a *context*, and then using that for creating sockets. In C, it's the `zmq_ctx_new()` 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 ZeroMQ instances. If that's explicitly what you want, OK, but otherwise remember:

Call `zmq_ctx_new()` once at the start of a process, and `zmq_ctx_destroy()` once at the end.

If you're using the `fork()` system call, do `zmq_ctx_new()` *after* the fork and at the beginning of the child process code. In general, you want to do interesting (ZeroMQ) stuff in the children, and boring process management in the parent.

Making a Clean Exit

top prev next

Classy programmers share the same motto as classy hit men: always clean-up when you finish the job. When you use ZeroMQ 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 else you get memory leaks, unstable applications, and generally bad karma.

Memory leaks are one thing, but ZeroMQ 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()` function will hang forever. And even if you close all sockets, `zmq_ctx_destroy()` 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 ZeroMQ objects we need to worry about are messages, sockets, and contexts. Luckily it's quite simple, at least in simple programs:

- Use `zmq_send()` and `zmq_recv()` when you can, as it avoids the need to work with zmq_msg_t objects.
- If you do use `zmq_msg_recv()`, always release the received message as soon as you're done with it, by calling `zmq_msg_close()`.
- If you are opening and closing a lot of sockets, that's probably a sign that you need to redesign your application. In some cases socket handles won't be freed until you destroy the context.
- When you exit the program, close your sockets and then call `zmq_ctx_destroy()`. This destroys the context.

This is at least the case for C development. In a language with automatic object destruction, sockets and contexts will be destroyed as you leave the scope. If you use exceptions you'll have to do the clean-up in something like a "final" block, the same as for any resource.

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, try to run before you can safely walk, below is the quick and dirty guide to making a clean exit in a *multithreaded* ZeroMQ application.

First, do not try to use the same socket from multiple threads. Please 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), and 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.

Why We Needed ZeroMQ

top prev next

Now that you've seen ZeroMQ 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 of these:

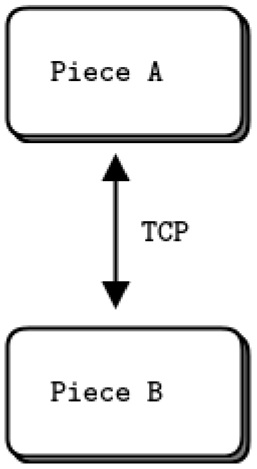
- How do we handle I/O? Does our application block, or do we handle I/O in the background? This is a key design decision. Blocking I/O creates architectures that do not scale well. But background I/O can be very hard to do right.
- How do we handle dynamic components, i.e., pieces that go away temporarily?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

Take a typical open source project like Hadoop Zookeeper and read the C API code in `src/c/src/zookeeper.c`. When I read this code, in January 2013, it was 4,200 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.

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

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

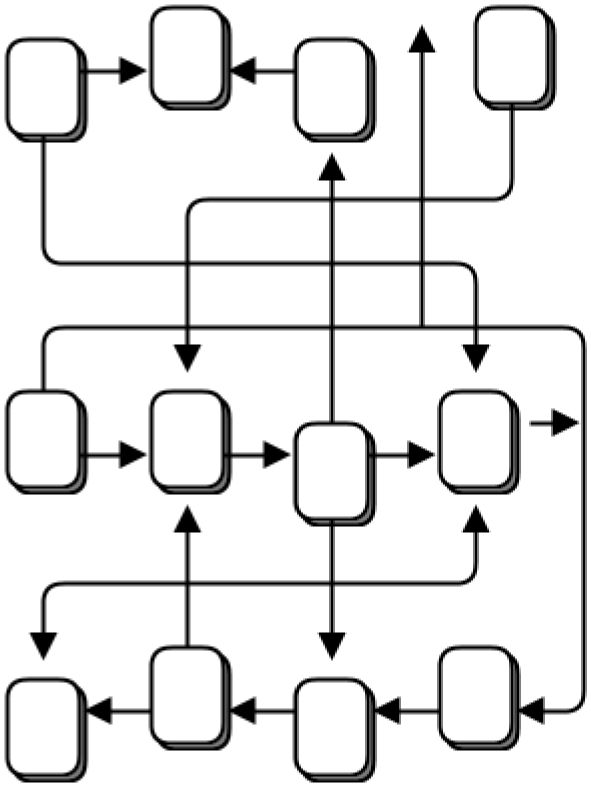
Figure 7 - Messaging as it Starts



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 allows applications to 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, and fourth broker and make some failover scheme. People do this. It creates more moving pieces, more complexity, and more things to break.

And a broker-centric setup 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.

Figure 8 - Messaging as it Becomes



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.

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 which you just link, 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 ZeroMQ: 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 ZeroMQ applications need no locks, semaphores, or other wait states.
- Components can come and go dynamically and ZeroMQ 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, ZeroMQ 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 pub-sub. 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 from zero to gigabytes large. When you want to represent data you choose some other product on top, such as msgpack, Google's protocol buffers, and others.
- It handles network errors intelligently, by retrying automatically in cases where it makes sense.
- 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 ZeroMQ.

Actually ZeroMQ 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_recv()` and `zmq_send()`. 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 nodes on one network (node is a box)—it's all the same, with no application code changes.

Socket Scalability

top prev next

Let's see ZeroMQ'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 the `top` command', 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 ZeroMQ 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.

Upgrading from ZeroMQ v2.2 to ZeroMQ v3.2

top prev next

Compatible Changes

top prev next

These changes don't impact existing application code directly:

- Pub-sub filtering is now done at the publisher side instead of subscriber side. This improves performance significantly in many pub-sub use cases. You can mix v3.2 and v2.1/v2.2 publishers and subscribers safely.
- ZeroMQ v3.2 has many new API methods (`zmq_disconnect()`, `zmq_unbind()`, `zmq_monitor()`, `zmq_ctx_set()`, etc.)

Incompatible Changes

top prev next

These are the main areas of impact on applications and language bindings:

- Changed send/recv methods: `zmq_send()` and `zmq_recv()` have a different, simpler interface, and the old functionality is now provided by `zmq_msg_send()` and `zmq_msg_recv()`. Symptom: compile errors. Solution: fix up your code.
- These two methods return positive values on success, and -1 on error. In v2.x they always returned zero on success. Symptom: apparent errors when things actually work fine. Solution: test strictly for return code = -1, not non-zero.
- `zmq_poll()` now waits for milliseconds, not microseconds. Symptom: application stops responding (in fact responds 1000 times slower). Solution: use the `ZMQ_POLL_MSEC` macro defined below, in all `zmq_poll` calls.
- `ZMQ_NOBLOCK` is now called `ZMQ_DONTWAIT`. Symptom: compile failures on the `ZMQ_NOBLOCK` macro.
- The `ZMQ_HWM` socket option is now broken into `ZMQ_SNDHWM` and `ZMQ_RCVHWM`. Symptom: compile failures on the `ZMQ_HWM` macro.
- Most but not all `zmq_getsockopt()` options are now integer values. Symptom: runtime error returns on `zmq_setsockopt` and `zmq_getsockopt`.
- The `ZMQ_SWAP` option has been removed. Symptom: compile failures on `ZMQ_SWAP`. Solution: redesign any code that uses this functionality.

Suggested Shim Macros

top prev next

For applications that want to run on both v2.x and v3.2, such as language bindings, our advice is to emulate v3.2 as far as possible. Here are C macro definitions that help your C/C++ code to work across both versions (taken from CZMQ):

```
#ifndef ZMQ_DONTWAIT
#     define ZMQ_DONTWAIT       ZMQ_NOBLOCK
```

```
#endif
#if ZMQ_VERSION_MAJOR == 2
#     define zmq_msg_send(msg,sock,opt) zmq_send (sock, msg, opt)
#     define zmq_msg_recv(msg,sock,opt) zmq_recv (sock, msg, opt)
#     define zmq_ctx_destroy(context) zmq_term(context)
#     define ZMQ_POLL_MSEC      1000            //  zmq_poll is usec
#     define ZMQ_SNDHWM ZMQ_HWM
#     define ZMQ_RCVHWM ZMQ_HWM
#elif ZMQ_VERSION_MAJOR == 3
#     define ZMQ_POLL_MSEC      1               //  zmq_poll is msec
#endif
```

Warning: Unstable Paradigms!

top prev next

Traditional network programming is built on the general assumption that one socket talks to one connection, one peer. There are multicast protocols, but 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 ZeroMQ 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 ZeroMQ's scalability.

This is 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 ZeroMQ scales 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 ZeroMQ code for a day or two (and maybe three or four days), you may feel confused, especially by how simple ZeroMQ makes things for you, and you may try to impose that general assumption on ZeroMQ, 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.