Practical Perl Tools
My Hero, Zero (Part 1)

DAVID N. BLANK-EDELMAN

David N. Blank-Edelman is the director of technology at the Northeastern University College of Computer and Information Science and the author of the O'Reilly book *Automating System Administration with Perl* (the second edition of the Otter book), available at purveyors of fine dead trees everywhere. He has spent the past 24+ years as a system/network administrator in large multi-platform environments, including Brandeis University, Cambridge Technology Group, and the MIT Media Laboratory. He was the program chair of the LISA '05 conference and one of the LISA '06 Invited Talks co-chairs. David is honored to have been the recipient of the 2009 SAGE Outstanding Achievement Award and to serve on the USENIX Board of Directors beginning in June of 2010. dnb@ccs.neu.edu

In the last column, I spent some time exploring MongoDB, a database that challenges in some respects what it means to be a database. For this column, I'd like to take a look at a message queuing system that does the same for message queuing systems: ZeroMQ (written most often as 0MQ. Even though I'm not that hip, I'll use that representation most of the time below).

If the term "message queuing system" makes you feel all stifle-a-yawn enterprise-y, boring business service bus-ish, get-off-my-lawn-you-kids, we were doing that in the '80s-like, then I would recommend taking another look at how serious systems are getting built these days. If you are like me, you will notice again and again places in which tools are adopting message-bus architectures where you might not expect them. These architectures turn out to be an excellent way to handle the new reality of distributed systems, such as those you might find when you've launched into your favorite cloud provider. This is why message queuing systems are at the heart of packages like MCollective and Sensu. They often allow you to build loosely coupled and dynamic systems more easily than some traditional models.

Our friend Wikipedia talks about message queues as "software-engineering components used for interprocess communication, or for inter-thread communication within the same process.... Message queues provide an asynchronous communications protocol, meaning that the sender and receiver of the message do not need to interact with the message queue at the same time." Message queuing systems like ActiveMQ and RabbitMQ let you set up message broker servers so that clients can receive or exchange messages.

Now, back to the MongoDB comparison: Despite having MQ at the end of the name like ActiveMQ and RabbitMQ, ZeroMQ is a very different animal from the other MQs. I don't think I can do a better job setting up how it is different than by quoting the beginning of the official 0MQ Guide:

> ØMQ (also known as ZeroMQ, 0MQ, or 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 fan-out, pub-sub, task distribution, and request-reply. It's fast enough to be the fabric for clustered products. Its asynchronous I/O model gives you scalable multicore applications, built as asynchronous message-processing tasks. It has a score of language APIs and runs on most operating systems. ØMQ is from iMatix and is LGPLv3 open source.

Let me emphasize a key part of the paragraph above. With 0MQ, you don't set up a distinct 0MQ message broker server like you might with a traditional MQ system. There's no zeromq binary, there's no /etc/init.d/zeromq, no /etc/zeromq for config files, no ZeroMQ Windows service to launch (or whatever else you think about when bringing up a server). Instead, you use the 0MQ libraries to add magic to your programs. This magic makes building your own message-passing architecture (whether it's hub and spoke, mesh, pipeline, etc.) a lot easier than you might expect. It takes a lot of the pain out of writing clients, servers, peers, and so on. I'll show exactly what this means in a moment.

I want to note one more thing before actually diving into the code. ZeroMQ looks really basic at first, probably because the network socket model *is* pretty basic. But, like anyone who has built something with a larger-than-usual construction set, such as a ton of Tinkertoys, Legos, or maybe a huge erector set (if you are as old as dirt), at a certain point you step back from a creation that has somehow grown taller than you are and say "whoa." I know I had this experience reading the ZeroMQ book (disclosure: published by O'Reilly, who also published my book). In this book, which I highly recommend if ZeroMQ interests you at all, it describes a ton of different patterns that are essentially building blocks. At some point, you'll have that "whoa" moment when you realize that these building blocks offer everything you need to construct the most elaborate or elegant architecture your heart desires. Given the length of this column, I'll only be able to look at the simplest of topologies, but I hope you'll get a hint of what's possible.

Socket to Me

Okay, that's probably the single most painful heading I've written for this column. Let's pretend it didn't happen.

Everything 0MQ-based starts with the basic network socket model, so I'll attempt a three-sentence review (at least the parts that are relevant to 0MQ). Sockets are like phone calls (and indeed the combination of an IP address and a port attempts to provide a unique phone number). To receive a connection, you create a socket and then bind to it to listen for connections (I'm leaving out accept() and a whole Stevens book, but bear with me); to make a connection, you connect() to a socket already set to receive connections. Receiving data from an incoming connection is done via a recv()-like call; sending data to the other end is performed with a send()-like call.

Those three sentences provide the key info you need to know to get started with 0MQ socket programming. Like one of those late-night commercials for medications, I feel I should tag on a whole bunch of disclaimers, modifiers, and other small print left out of the commercial, but I'm going to refrain except for these two things:

- There are lots of nitty-gritty details to (good) socket programming I'm not even going to touch. 0MQ handles a bunch of that for you, but if you aren't going to use 0MQ for some reason, be sure to check out both the non-Perl references (e.g., the Stevens books) and the Perl-related references (Lincoln Stein wrote a great book called *Network Programming with Perl* many eons ago).
- If you do want to do plain ol' socket programming in Perl, you'll want to use one of the socket-related Perl modules to make the job easier. Even though Socket.pm ships with Perl, I think you'll find IO::Socket more pleasant to use. With these modules, you can open up a socket that connects to another host easily and print() to it as if it were any other file handle. Again, this is just a "by the way" sort of thing since we're about to strap on a rocket motor and use 0MQ sockets from Perl.

To get started with 0MQ in Perl, you'll need to pick a Perl module that provides an interface to the 0MQ libraries. At the moment, there are two choices for modules worth considering, plus or minus one: ZMQ::LibZMQ3 and ZMQ::FFI. The former is the successor to what used to be called just ZeroMQ. The author of that module decided to create modules specifically targeted to specific major versions of the 0MQ libraries (v2 and v3). It offers an API that is very close to the native library (as a quick aside, the author also provides a ZMQ module which will call ZMQ::LibZMQ3 or ZMQ::LibZMQ2, but the author suggests in most cases to use the version-specific library directly, hence my statement of "plus or minus one," above). The ZMQ::FFI uses libffi to provide a slightly more abstract interface to 0MQ that isn't as 0MQ-version specific. (In case you are curious about FFI, its docs say, "FFI stands for Foreign Function Interface. A foreign function interface is the popular name for the interface that allows code written in one language to call code written in another language.")

In this column, I'll use ZMQ::LibZMQ3 because it allows me to write code that looks like the C sample code provided in the 0MQ user guide (and in the 0MQ book). For the sample code in this part of a two-part column, I'll keep things dull and create a simple echo client-server pair. The server will listen for incoming connections and echo back any messages the connected clients sent to it. First, I'll look at the server code, because it's going to give me a whole bunch of things to talk about:

```
use ZMQ::LibZMQ3;
use ZMQ::Constants qw(ZMQ_REP);

my $ctxt = zmq_init;
my $socket = zmq_socket( $ctxt, ZMQ_REP );

my $rv = zmq_bind( $socket, "tcp://127.0.0.1:8888" );

while (1) {
  my $msg = zmq_recvmsg($socket);
  print "Server received:" . zmq_msg_data($msg) . "\n";
  my $msg = zmq_msg_init_data( zmq_msg_data($msg) );
  zmq_sendmsg( $socket, $msg );
}
```

The first thing to note about this code is that loads both the 0MQ library module and a separate constants module. Depending on how you installed the library module, the constants module likely was installed at the same time for you. This module holds the definition for all of the constants and flags you'll be using with 0MQ. There are a bunch—you'll see the first one in just a couple of lines. The next line of code initializes the 0MQ

environment, something you have to do before you use any other 0MQ calls. 0MQ contexts usually don't have to come into play except as a background thing unless you are doing some fairly complex programming, so I'm not going to say more about them here.

With all of that out of the way, it is time to create your first socket. Sockets get created in a context and are of a specific type (in this case ZMQ_REP, or just REP). Socket types are a very important concept in 0MQ, so I'll take a quick moment away from the code to discuss them.

I don't know the last time you played with Tinkertoys but you might recall that they consisted of a few basic connector shapes (square, circle, etc.), which accepted rods at specific angles. If you wanted to build a cube, you had to use the connectors that had holes at 90-degree angles and so on. With 0MQ, specific socket types get used for creating certain sorts of architectures. You can mix and match to a certain extent, but some combinations are more frequently used or more functional than others.

For example, in the code I am describing, I will use REQ and REP sockets (REQ for synchronously sending a REQuest, REP for synchronously providing a REPly). You will see a similar pair in an example in the next issue. You might be curious what the difference is between a REQ and REP socket because as Dr. Seuss might say, "a socket's a socket, no matter how small." The short answer is the different socket types do slightly different things around message handling (e.g., how message frames are constructed, etc.). See the 0MQ book for more details.

Now that you have a socket constructed, you can tell it to listen for connections, which is done by performing a zmq_bind(). Here you can see that I have asked to listen to unicast TCP/IP connections on port 8888 of the local host. 0MQ also knows how to handle multicast, inter-process, and inter-thread connections. At this point, you are ready to go into a loop that will listen for messages. The zmq_recvmsg() blocks until it receives an incoming message. It returns a message object, the contents of which are displayed using zmq_msg_data($msg). To reply to the message you just received, you construct a message object with the contents of the message you received and send it back over the socket via zmq_sendmsg($socket, $msg). That's all you need to construct a simple server; now, I'll look at the client code:

```
use ZMQ::LibZMQ3;
use ZMQ::Constants qw(ZMQ_REQ);

my $ctxt = zmq_init;
my $socket = zmq_socket( $ctxt, ZMQ_REQ );

zmq_connect( $socket, "tcp://127.0.0.1:8888" );

my $counter = 1;

print "I am $$\n";

while (1) {
  my $msg = zmq_msg_init_data("$counter:$$");
  zmq_sendmsg( $socket, $msg );
  print "Client sent message " . $counter++ . "\n";
  my $msg = zmq_recvmsg($socket);
  print "Client received ack:" . zmq_msg_data($msg) . "\n";
  sleep 1;
}
```

The client code starts out almost identically (note: I said almost, the socket type is different. I once spent a very frustrating hour trying to debug a 0MQ program because I had written ZMQ_REP instead of ZMQ_REQ in one place in the code). It starts to diverge from the server code because, instead of listening for a connection, it is set to initiate one by using zmq_connect(). In the loop, you construct a simple message (the message number plus the PID of the script that is running) and send it on the socket. Once you've sent the message, you wait for a response back from the server using the same exact code the server used to receive a message. REQ-REP sockets are engineered with the expectation that a request is made and a reply is returned, so you really do want to send a reply back. Finally, I should mention there is a sleep statement at the end of this loop just to keep things from scrolling by too quickly (0MQ is FAST), but you can feel free to take it out.

Let's take the code for a spin. If you start up the server, it sits and waits for connections:

```
$ ./zmqserv1.pl
```

In a separate window, start the client:

Immediately, the client prints something like:

```
I am 86989
Client sent message 1
Client received ack:1:86989
Client sent message 2
Client received ack:2:86989
Client sent message 3
Client received ack:3:86989
…
```

and the server also shows this:

```
Server received:1:86989
Server received:2:86989
Server received:3:86989
…
```

Now, you can begin to see what all of the fuss is about with 0MQ. First, if you stop the server and then start it again (while leaving the client running):

```
$ ./zmqserv1.pl
Server received:1:87110
Server received:2:87110
Server received:3:87110
^C
$ ./zmqserv1.pl
Server received:4:87110
Server received:5:87110
Server received:6:87110
Server received:7:87110
^C
$ ./zmqserv1.pl
Server received:8:87110
Server received:9:87110
Server received:10:87110
^C
```

During all of this, the client just said:

```
$ ./zmqcli1.pl
I am 87110 Client sent message 1
Client received ack:1:87110
Client sent message 2
Client received ack:2:87110
Client sent message 3
Client received ack:3:87110
Client sent message 4
Client received ack:4:87110
Client sent message 5
Client received ack:5:87110
Client sent message 6
Client received ack:6:87110
Client sent message 7
Client received ack:7:87110
Client sent message 8
Client received ack:8:87110
Client sent message 9
Client received ack:9:87110
Client sent message 10
Client received ack:10:87110
```

Here you are seeing 0MQ's sockets auto-reconnect (and do a little bit of buffering). This isn't the only magic going on behind the scenes, but it is the easiest to demonstrate.

Now, I'll make the topology a little more interesting. Suppose you want to have a single server with multiple clients connected to it at the same time. Here's the change you'd have to make to the server code:

(nada)

And the change to the client code:

(also nada)

All you have to do is start the server and spin up as many copies of the client as you desire. Let's start up five clients at once:

```
$ for ((i=0;i<5;i++)); do ./zmqcli1.pl & done
```

On the client side, you see a mishmash of the outputs from the client (that eventually return to lockstep):

```
I am 87242
I am 87241
Client sent message 1 I am 87239
Client sent message 1 I am 87243
Client sent message 1
Client received ack:1:87241
I am 87240
Client received ack:1:87243
Client sent message 1 Client sent message 1
Client received ack:1:87239
Client received ack:1:87242
Client received ack:1:87240
Client sent message 2
Client sent message 2
Client sent message 2
Client sent message 2
Client sent message 2
Client received ack:2:87239
Client received ack:2:87241
Client received ack:2:87243
Client received ack:2:87240
Client received ack:2:87242
Client sent message 3
Client sent message 3
Client sent message 3
Client sent message 3
Client sent message 3
Client received ack:3:87241
Client received ack:3:87239
Client received ack:3:87240
Client received ack:3:87243
Client received ack:3:87242
```

On the server side, the output is a little more orderly:

```
$ ./zmqserv1.pl
Server received:1:87241
Server received:1:87239
Server received:1:87243
Server received:1:87242
Server received:1:87240
Server received:2:87239
Server received:2:87241
Server received:2:87243
Server received:2:87242
Server received:2:87240
```

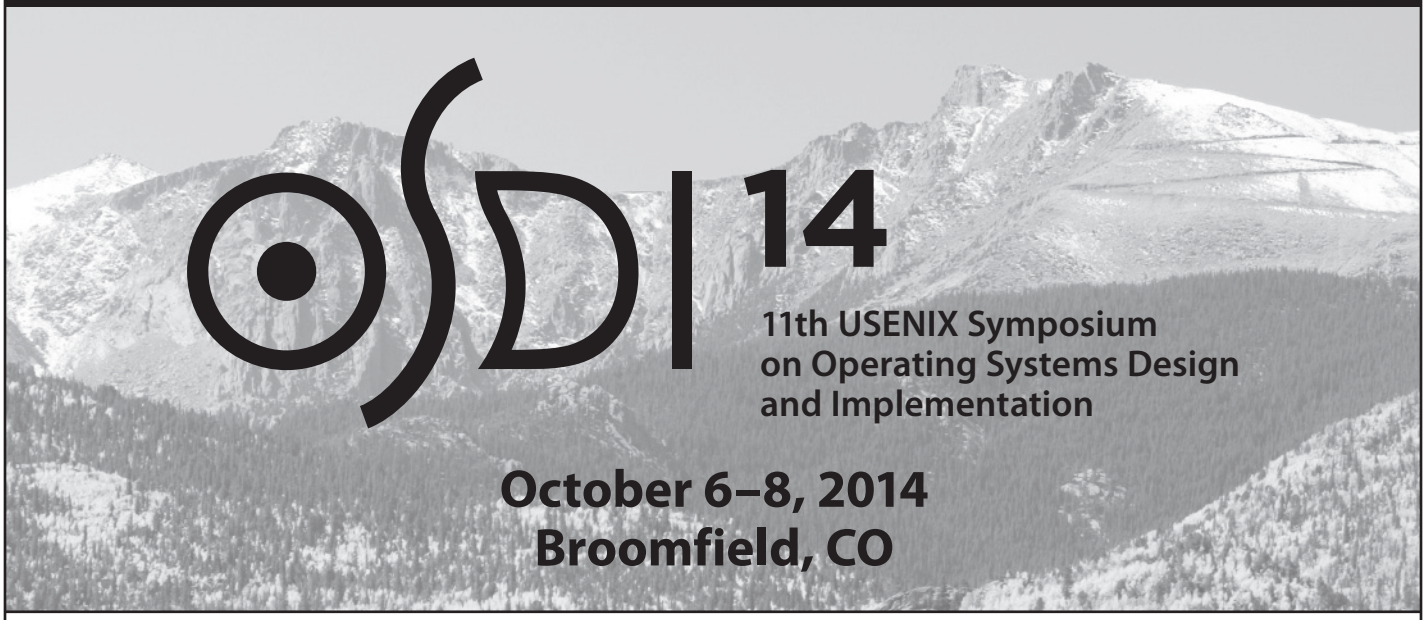
```
Server received:3:87241
Server received:3:87239
Server received:3:87240
Server received:3:87243
Server received:3:87242
```

It is pretty cool that the server was able to handle multiple simultaneous connections without any code changes, but one thing that may not be clear is that 0MQ is not only handling these connections, it's also automatically load-balancing between them as well. That's the sort of behind-the-scenes magic I think is really awesome.

I believe it is always good to end a show after a good magic trick, so I'll wind up part 1 of this column right here. Join me next time when I will look at other ZeroMQ socket types and build some even cooler network topologies.

Take care, and I'll see you next time.

SAVE THE DATE!



Join us in Broomfield, CO, October 6–8, 2014, for the **11th USENIX Symposium on Operating Systems Design and Implementation (OSDI '14).** The Symposium brings together professionals from academic and industrial backgrounds in what has become a premier forum for discussing the design, implementation, and implications of systems software.

Don't miss the co-located workshops on Sunday, October 5

Diversity '14: 2014 Workshop on Supporting Diversity in Systems Research

HotDep '14: 10th Workshop on Hot Topics in Dependable Systems

HotPower '14: 6th Workshop on Power-Aware Computing and Systems

INFLOW '14: 2nd Workshop on Interactions of NVM/Flash with Operating Systems and Workloads

TRIOS '14: 2014 Conference on Timely Results in Operating Systems

All events will take place at the Omni Interlocken Resort

usenix
THE ADVANCED
COMPUTING SYSTEMS
ASSOCIATION

www.usenix.org/osdi14