Practical Perl Tools
Zero Plus One

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 our last time together, we spent the column exploring ZeroMQ (sometimes written 0MQ). We looked at the basics of what it is, why it is cooler than I will ever be, and began to look at some sample Perl code that uses it. With this column, I hope to take the subject just a little further by exploring a few slightly more complex 0MQ topics. I'm going to (mostly) avoid rehashing the basics from last time, so I recommend you check out that column first.

Begin Again, Again

Early in the last column, I mentioned that there were two (well, two and a half) main strains of Perl modules that would let you work with ZeroMQ from Perl. The first was the ZMQ::LibZMQx series (ZMQ::LibZMQ2 and ZMQ::LibZMQ3 for versions 2 and 3 of the ZeroMQ libraries). There was also a thin wrapper around these (hence the "half" comment) called ZMQ that would call one of the two. The second main strain I mentioned was ZMQ::FFI, which used the libffi library as a bridge to the ZeroMQ libraries. In the last column, I made the decision to show code using the first kind of module (ZMQ::LibZMQ3). I still think this is a fine and dandy sort of thing to do; but, since that column was written, it has become more apparent to me that the ZMQ::FFI may be the future of ZeroMQ for Perl. For example, although there is not yet a ZMQ::LibZMQ4 to use version 4 of ZeroMQ (and I'm not sure there will be), ZMQ::FFI works with it out of the box. This may be important to you if, for example, you were interested in using some of the brand-new encrypted transport hotness that just arrived in version 4 of ZeroMQ. The version 3 libraries work perfectly fine, so if you are already happily using ZMQ::LibZMQ3 there's no real need to rush out right this minute and rewrite all of your code.

If indeed ZMQ::FFI will ascend as the preferred module, I think it would be a service to you, dear reader, that I demonstrate how to code using it. So, in this column, we'll be switching horses midstream and will start using it. To help ease the transition a little bit, I'll take a look at the example client-server (or REQ-REP) code that I ended the previous column with—this time coded using ZMQ::FFI. Here's the server version:

```
use ZMQFFI;
use ZMQFFIConstants qw(ZMQ_REP);

my $ctxt   = ZMQ::FFI->new();
my $socket = $ctxt->socket(ZMQ_REP);

my $rv = $socket->bind('tcp://127.0.0.1:8888');

while (1) {
    my $msg = $socket->recv();
    print "Server received: $msg\n";

    $socket->send($msg);

}
```

The differences between this version and the ZMQ::LibZMQ3 are pretty small but worth pointing out. First, this version is more object-oriented. Everything except the messages being received and sent are objects; whereas before we would just "zmq_recvmsg($socket);" here you can see that our recv() is a method of the socket object. The second significant difference is that we no longer have to do any special processing to prepare or receive messages. This simplifies the code a wee bit.

The client code has almost identical changes from the previous version:

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

my $ctxt   = ZMQ::FFI->new();
my $socket = $ctxt->socket(ZMQ_REQ);

$socket->connect('tcp://127.0.0.1:8888');

my $counter = 1;

print "I am $$\n";

while (1) {
    $socket->send("$counter:$$");
    print "Client sent message " . $counter++ . "\n";

    my $msg = $socket->recv();
    print "Client received ack:$msg\n";
    sleep 1;
}
```

I know I mentioned this last time, but as I look over this code now I feel compelled once again to offer my appreciation to the 0MQ folks and the author of this module for providing a framework that lets us write code that looks this simple but does so much great stuff behind the scenes. Let's take this code a little further.

Ah, to Be Young and Asynchronous

I'm not sure whether you recall the output of the previous examples, but a key quality of the code it demonstrated was the synchronous nature of the REQ-REP (request-reply) socket types. They are constructed in such a way as to mandate a strict "you request - I reply, you request - I reply" pattern. It doesn't work to fire off a bunch of requests without recv()-ing the replies back for each one. If we want to do that sort of thing, we need to explore some more complicated socket types. There are two sets of socket types that can help with this. The first is an extension of the REQ-REP types we've been using. The second introduces a different paradigm, so we'll hold off on that until the end of this column when you'll see a second way to program with ZeroMQ.

The two new socket types I'd like to introduce you to now are DEALER and ROUTER (ZMQ_DEALER and ZMQ_ROUTER). I find it easiest to remember them by matching them up to the previous types we've been using. A DEALER is like a REQ socket in that it gets used to connect in a "client-like" way to other socket types (and, indeed, those types could be the REP type we've seen before). If you can imagine a card dealer sitting in the middle of a table, dealing out messages to eager card players who receive them, that's the basic idea. I call this "client-like" because the basic direction of packets is out from the DEALER to a receiving (server-esque) socket. The flip side to a DEALER socket is a ROUTER socket. ROUTER sockets are expected to receive messages from a number of sources (which could be REQ sockets or DEALERs). If you think of a network router that sits on a network waiting to receive packets that it passes along, you've got the right idea again.

I'll start off checking out what happens if I combine a REQ and a ROUTER socket. I'll use almost exactly the same REQ code as before with a few small twists:

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

my $ctxt   = ZMQ::FFI->new();
my $socket = $ctxt->socket(ZMQ_REQ);

$socket->connect('tcp://127.0.0.1:8888');

my $counter = 1;

print "I am $$\n";

while (1) {
    $socket->send("$counter:$$");
    print "Client sent message " . $counter++ . "\n";

    my @msgs = $socket->recv();
    print "Client received ack:$msgs[0]\n";
    sleep 1;
}
```

The only change I want to bring to your attention is the retrieval of an array of values from recv(). I can best explain the recv() change in the context of the second piece of code—the one with a ROUTER socket:

```
use ZMQ::FFI;
use ZMQ::FFI::Constants qw(ZMQ_ROUTER);

my $ctxt   = ZMQ::FFI->new();
my $socket = $ctxt->socket(ZMQ_ROUTER);

my $rv = $socket->bind('tcp://127.0.0.1:8888');

while (1) {

    my ( $id, $spacer, $msg ) = $socket->recv_multipart();
    print "Server received: $msg\n";

    $socket->send_multipart( [ $id, '', $msg ] );
}
```

Now we see an interesting change from the REP code we looked at before. In this code, I've replaced recv() with recv_multipart and send() with send_multipart(). The distinction is the _multipart forms know how to handle the sending and receipt of multiple message frames at the same time.

Let me step back a moment to explain why this is necessary. Behind the scenes, messages actually consist of multiple frames. With the REQ->REP code, we didn't have to care because there was a one-to-one relationship between a message and its reply that ZeroMQ handled automatically. Even with multiple REQ clients talking to the same REP "server," this relationship was always true (i.e., the expectation is a REP would reply to the client before moving on). With a REQ-REP combination, ZeroMQ handled details like "Where do I send this reply back to?" for us.

With a ROUTER socket, we don't have the promise of a synchronous traffic pattern. We could get a message and choose to reply to it or perhaps forward that message along to another socket. To make this flexibility possible, ZeroMQ has to have a way for the application using a ROUTER socket to know where the message is coming from so that it can either reply to the source or pass the info on so that someone else can. This is implemented in a super simple fashion: Messages consist of multiple frames that have a source identifier frame, message contents frames, and a blank frame in between these two things. In the code above, recv_multipart returns these three parts.

Now, the fact that the REQ socket is talking to a ROUTER doesn't change the REQ socket's synchronous nature. A REQ socket always needs to get a response back to a message it sends. We accommodate this need by having our ROUTER code echo back the message it received. By sending a message using the same three frames (via send_multipart()), the message is destined for the client that made the request and everybody is happy.

I'll run this code and see what happens. First, I'll start the ROUTER-based server and then spin up three REQ-based clients simultaneously. Here's some of the output from the server:

```
Server received: 1:72551
Server received: 1:72550
Server received: 1:72549
Server received: 2:72551
Server received: 2:72549
Server received: 2:72550
Server received: 3:72549
Server received: 4:72549
Server received: 3:72551
Server received: 4:72551
Server received: 3:72550
Server received: 4:72550
Server received: 5:72549
Server received: 5:72551
Server received: 6:72551
Server received: 5:72550
Server received: 6:72550
Server received: 6:72549
Server received: 7:72549
```

This output shows each message the server receives. The first number is the request number the client sends, the second is what each client is calling itself (it's that client's PID). It is not exactly easy to tell that the ROUTER socket is behaving asynchronously, but it is.

If we wanted to go in the other direction and have a DEALER talk to a REP socket, that approach works swimmingly as well. As the ZeroMQ handbook says (referring to a previous REQ-REP example): "This gives us an asynchronous client that can talk to multiple REP servers. If we rewrote the 'Hello World' client using DEALER, we'd be able to send off any number of 'Hello' requests without waiting for replies." As in the previous example, we need to think a bit harder about actual message frames. The message our DEALER sends out should mimic the same format a REQ socket might send. This is spelled out more explicitly in the doc than I want to get into here; but, in short, that's sending out a spacer frame followed by the message contents in a separate frame. The tools are exactly the same as in the previous example (send_multipart()), so I won't repeat all of that code.

I'd like to show another architecture before this column is done, but I suspect you might be curious about the combination we haven't discussed, namely DEALER-ROUTER combinations. Not to be too glib but, yup, that works great, too. If you pair two asynchronous socket types, you can make spiffy things like a component that can sit in between front-end and back-end pieces, asynchronously receiving messages and fanning them out as desired.

PUB and SUB

The final socket types we'll look at provide a slightly different paradigm than the ones we've seen so far. This paradigm came up when I talked about Redis a few columns ago and is becoming more common these days. With the publisher-subscriber (pub-sub) paradigm, there is a component that sends out messages (the publisher) that get "tagged" as being associated with specific topics. Some number of clients can then connect to the publisher, indicate interest in one or more of those topics, and then receive just the messages tagged with those topics.

I'll let some code do the introduction. Here's the publisher (the server that clients will connect to in order to receive their content):

```
use ZMQ::FFI;
use ZMQ::FFI::Constants qw(ZMQ_PUB);

my $ctxt   = ZMQ::FFI->new();
my $socket = $ctxt->socket(ZMQ_PUB);

my $rv = $socket->bind('tcp://127.0.0.1:8888');

# bogus way to sync
sleep 5;

while (1) {

    $socket->send('bunnies furry');
    $socket->send('doggies barky');
    $socket->send('fishies swimmy');

    sleep 1;
}
```

This should look remarkably like the other examples, because, well, it is. The only difference is that we've changed the socket type. I consider the lack of difference to be a plus because it demonstrates how it is easy to build on prior knowledge when working with ZeroMQ. Before turning to the slightly more interesting "client," I do want to briefly discuss the comment above about the sleep() call being a bogus way to sync.

When dealing with a publisher-subscriber model, one of the common problems a first-time coder encounters is a synchronization problem. If the publisher publishes information before all of the subscribers have connected, the latecomers will miss messages. This isn't always a problem (e.g., the above code which repeats until interrupted), but in most real-world cases, it's highly suboptimal. There are a number of ways to fix this; the one above, where we just wait a bit, is probably the worst.

Far preferable would be for the publisher to have some way to know when all of the subscribers are present and listening. If you know how many subscribers constitutes "all," the easiest way is to dedicate a second socket pair in your code to just this sort of out-of-band signaling. You could imagine using a REQ-REP socket pair where the subscriber checks in to the publisher by sending something over the REQ, and the publisher acks the notice on the REP (basically using the code we've seen before). Once the publisher is satisfied everyone is tuned in, it can then begin sending actual content.

Now let's look at a simple subscriber:

```
use ZMQ::FFI;
use ZMQ::FFI::Constants qw(ZMQ_SUB);

my $ctxt   = ZMQ::FFI->new();
my $socket = $ctxt->socket(ZMQ_SUB);

$socket->connect('tcp://127.0.0.1:8888');

my @topics = qw(bunnies doggies fishies);

# subscribe to a random topic
my $interest = $topics[ int( rand(3) ) ];
$socket->subscribe($interest);

while (1) {

    my ( $topic, $message ) = split / /, $socket->recv();
    print "Today I learned that $topic are $message!\n";

}
```

Again, super simple. The code picks a random topic and registers interest in this topic via a subscribe() call. From that point on, it will only "hear" messages on that topic when it calls recv(), like so:

```
Today I learned that bunnies are furry!
Today I learned that bunnies are furry!
Today I learned that bunnies are furry!
Today I learned that bunnies are furry!
```

If I were to move the topic subscription code into the loop so it picks a random topic on the fly:

```
while (1){

    # subscribe to a random topic
    my $interest = $topics[ int( rand(3) ) ];
    $socket->subscribe($interest);

    my ( $topic, $message ) = split / /,$socket->recv();
    …
```

we get the expected results:

```
Today I learned that fishies are swimmy!
Today I learned that doggies are barky!
Today I learned that fishies are swimmy!
Today I learned that bunnies are furry!
Today I learned that doggies are barky!
```

I won't show you the server output (because you are probably fully up on your bunnies vs. doggies distinction), but we can easily run a metric ton of subscribers at the same time against our publisher. And, as before, no code changes are necessary to support going from one-to-one connections to multi-to-one.

Pretty cool, so with that, let's wrap. Take care, and I'll see you next time.