iVoyeur
Crosstalk

DAVE JOSEPHSEN

Dave Josephsen is the author of *Building a Monitoring Infrastructure with Nagios* (Prentice Hall PTR, 2007) and is senior systems engineer at DBG, Inc., where he maintains a gaggle of geographically dispersed server farms. He won LISA '04's Best Paper award for his co-authored work on spam mitigation, and he donates his spare time to the SourceMage GNU Linux Project.
dave-usenix@skeptech.org

The very moment that my three-year-old niece picked up the little scrap of PVC pipe, I knew what would happen. I saw in my mind's eye her baseball-bat swing into the back of her cousin's head, and knew it for the inevitable truth as predictably as the quadratic formula. As he lay crying on the floor it occurred to me that, being what we are, certain objects speak to us. Tools meet the hand and, for better or worse, beg to be used. They inspire us to action, but we need not learn to obey them; on the contrary, we spend years learning to resist.

Our primordial connection to tools is why there is a store in the mall where I can purchase a samurai sword, and it's why we say things like, "When all you have is a hammer, everything looks like a nail." I think it's something so deeply ingrained in us that it transcends physical objects. That may be presumptions of me, but personally, I often encounter software "solutions in want of a problem," by which I mean tools that are just so great I rack my brain trying to come up with a use for them.

So when I come across a quote like the following from Jon Gifford, "0MQ is unbelievably cool—if you haven't got a project that needs it, make one up" [1], I strongly relate. I even get excited. I can almost feel this 0MQ whatever-it-is grasped lightly in my hand, its weight as substantial as its balance is remarkable. It sings to me, and I realize it was made for me, and I for it; together, we are unstoppable. It shows me things: things that were, and are yet to be; things to smash; things, in fact, that cry out begging to be smashed. Together, we will drive elephants over the Alps, cross the Rubicon, defeat the infamous El Guapo and take back what is rightfully OURS.

You see how I get. Great tools inspire great deeds, especially if you can find a reason to use them, which, it turns out, is the case with 0MQ. But first things first; introductions are in order.

0MQ [2] (pronounced "Zero MQ") is, for lack of a better description, a sort of socket library. The goal of the project, briefly stated, is to "connect any code to any code, anywhere," and they're doing it by providing a simple, cross-platform, language-agnostic, asynchronous, messaging and concurrency protocol. 0MQ sockets automatically handle multiple simultaneous connections, provide fair queueing, and cooperate to form scalable multicore applications, using an interface you're already familiar with if you've done any socket programming. It is literally easier to use than it is to describe, but compared to traditional sockets there are two big differences.

First, compared to conventional sockets, which present a synchronous interface to either reliable, connection-oriented byte streams (SOCK_STREAM) or to unreliable, connection-less datagrams (SOCK_DGRAM), 0MQ sockets present an abstraction of an asynchronous message queue. The specific queueing behavior depends on the type of socket chosen by the programmer, but generally speaking, the programmer creates the socket, connects the socket, drops in a message, and that's it. There is no need to deal with the memory management associated with stream data on the client-end; 0MQ sockets know the size of the message and respond accordingly. Whereas conventional sockets transfer streams of bytes or datagrams, 0MQ sockets transparently transfer whole messages, quickly and safely.

Second, conventional sockets limit the programmer to one-to-one (two peers), many-to-one (many clients, one server), or in some cases one-to-many (multicast) relationships. Most 0MQ sockets (with one exception) can be connected to multiple endpoints, while simultaneously accepting incoming connections from multiple peers. Peer endpoints in either case may be other hosts on the network, other processes on the local machine, or other threads in the same process. The exact semantics of the message delivery depend on the type of socket chosen by the developer.

As I've said, there are several types of 0MQ sockets, each belonging to a "pattern," which corresponds to a style of distributed application framework. The available patterns are:

- Request-reply (classic TCP style client-server)
- Publish-subscribe (multicast, empowered, and simplified)
- Pipeline (think Hadoop style parallel application frameworks)
- Exclusive pair (inter-thread communication)

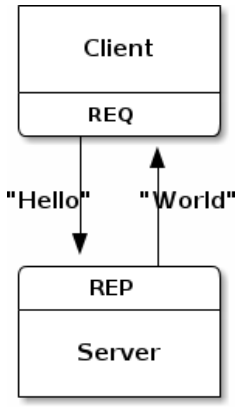

Figure 1: The 0MQ request-reply pattern

Sockets belonging to the same pattern are designed to work together to implement the pattern. The request-reply pattern, for example, is composed of a request socket and a reply socket. An application may use as many different patterns and socket types as it needs to accomplish its task, but socket types belonging to different patterns generally can't connect to each other directly. You cannot, for example, connect a PULL (pipeline) socket to a PUB (publish-subscribe) endpoint, but you could certainly write a single application that used both the publish-subscribe and pipeline patterns.

In this article, I'm only going to talk about two patterns, the request-reply, which is the most simplistic and therefore ideal for introductory examples, and the publish-subscribe pattern, which is the pattern that has me wanting to smash things.

The request-reply pattern, illustrated in Figure 1, is used for sending requests from a client to one or more instances of a service, and receiving subsequent replies to each request sent. Below is a Ruby script that implements a server on port 4242, which will wait for input from a 0MQ REQ (request) socket, and will reply with the word "foo":

```
require 'rubygems'
require 'ffi-rzmq'
context = ZMQ::Context.new(1)
socket = context.socket(ZMQ::REP)
socket.bind("tcp://*:4242")
```

```
while true do

request = ''
rc = socket.recv_string(request)
socket.send_string("foo")
end
```

In pseudo-code, the server ("responder" in 0MQ parlance):

- creates a socket of type REP (response);
- binds it to TCP port 4242 on 0.0.0.0/0 ;
- blocks waiting for a connection from a REQ (request socket);
- responds with the string "foo" once a connection is made (the message sent from the client is stored in the 'request' variable).

In the interest of word count, I'm going to keep the code listings to a minimum and just tell you that the client program does pretty much exactly the same thing in reverse; it:

- creates a socket of type REQ (request);
- connects it to TCP port 4242 on one or more servers;
- sends a message;
- blocks waiting for a response from the server.

The REQ and REP sockets work in lockstep; the client must send and then receive (in a loop, if necessary), and the server must receive and then send. Attempting any other sequence generates an error code. The server and client can start in any order, and 0MQ transparently handles multiple connections on either side, providing fair queueing if, for example, one client makes a connection every 500 milliseconds and another connects only once every two seconds. The developer is free to implement whatever protocol he likes over the connection, binary, or text; 0MQ does not know anything about the data you send except its size in bytes.

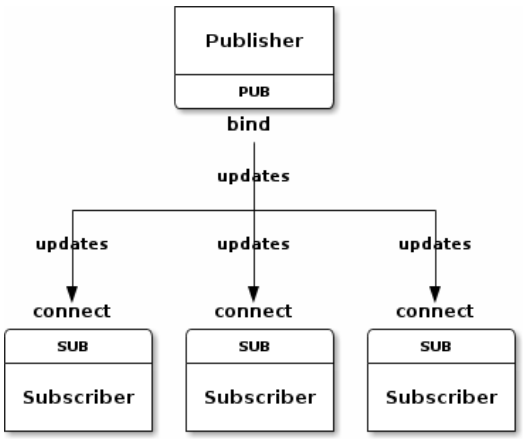

Figure 2: The 0MQ publish-subscribe pattern

The publish-subscribe pattern, depicted in Figure 2, is a multicast pattern, used for one-to-many distribution of data from a single publisher to multiple subscribers. A host wishing to distribute data creates a socket of type PUB, binds it to an address, and writes data to it as often as the application demands. A host wishing to receive data from the publisher creates a socket of type SUB, and connects it to the server. The PUB-SUB socket pair is asynchronous: the client receives in a loop while the server sends. Neither socket blocks waiting for the other.

Subscribers may connect to more than one publisher, using one "connect" call each time, and the received data will be interleaved on arrival so that no single publisher drowns out the others. A publisher that has no connected subscribers will simply /dev/null all its messages. Subscribers in a Pub-Sub relationship may specify one or more filters to control the content they're interested in receiving.

Now, if you're one of the four people who read this column with any regularity (hi Mom), I can safely assume that you have a large unruly gaggle of monitoring systems that you've managed to integrate. I can pretty safely make this assumption because the preponderant quantity of words I devote to this column that aren't dedicated to ranting or poorly disguised fart jokes are devoted to integrating monitoring tool A with monitoring tool B. That I cannot introduce a new tool without talking about how to integrate it with the other tools you're already using is testimony to an interesting fact; namely, that nobody uses a single tool for monitoring anymore.

Of course, "nobody" is a lot of somebodys. I'm certainly wrong there. Somebody somewhere probably uses one big tool for monitoring, but they're probably not reading a column that talks about integrating various tools all the time, unless they can't find a better source for ranting and poorly disguised fart jokes (unlikely). The rest of us are using a bunch of tools—this for systems availability, that for router metrics, the other for Web analytics . . . you get the point. The tools themselves are beginning to reflect this, as each monitoring tool slowly makes the realization that it isn't the center of the universe, that it is, in fact, a piece of a solution to a much larger need.

Five years ago we built tools like Cacti [3], which does everything from metrics polling, through data storage and into display—utterly self-contained. Today, we build tools like Graphite [4], a powerful display and analysis engine that's optimized for simplicity of input—built for integration. I gave up on Cacti because it's a dead end: you put SNMP data in, and there it stays forever. I love Graphite, because it takes any kind of data from any kind of system and is happy to share it everywhere. Graphite gives me a place to combine and compare the output of systems that cannot be made to talk to each other.

But what if we could go a few steps further. Imagine, every monitoring system—Ganglia [5], sflow [6], Nagios [7] , collectd [8], SNMPd [9], Graphite, etc.—publishing availability data and metrics using 0MQ Pub-Sub. All monitoring and metrics data available in a common syntax, using a common, lightweight, fast, message bus in a manner that didn't require a centralized server or single point of failure. Every agent could publish data to any collector that was interested all the time, and every collector could have access to any data it wanted.

Now there's a pipe to smack your cousin in the head with. The very fact of 0MQ's existence makes our lack of that pipe seem like hubris and stupidity, although some baby-steps have begun in the general direction I'm describing[10, 11]. I suspect the real challenge will be in coming up with a standard syntax for describing availability and metric data and getting a couple of the big players to adopt it. Maybe someone should hold a BoF.

Take it easy.

References

[1] http://www.slideshare.net/IanBarber/zeromq-is-the-answer-php-tek-11-version.

[2] http://www.zeromq.org/.

[3] http://www.cacti.net.

[4] http://graphite.wikidot.com/.

[5] http://ganglia.sourceforge.net/.

[6] http://www.sflow.org/.

[7] http://www.nagios.org.

[8] http://collectd.org/.

[9] http://net-snmp.sourceforge.net.

[10] https://github.com/mariussturm/nagios-zmq.

[11] https://github.com/jedi4ever/gmond-zmq.