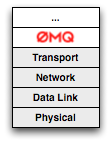
**ZeroMQ: Modern & Fast Networking Stack**

*By*[*Ilya Grigorik*](http://www.igvita.com/)*on****September 03, 2010***

[Berkeley Sockets](http://en.wikipedia.org/wiki/Berkeley_sockets) (BSD) are the de facto API for all network communication. With roots from the early 1980's, it is the original implementation of the TCP/IP suite, and arguably one of the most widely supported and critical components of any operating system today. BSD sockets that most of us are familiar with are **peer-to-peer connections**, which require explicit setup, teardown, choice of transport (TCP, UDP), error handling, and so on. And once you solve all of the above, then you are into the world of application protocols (ex: HTTP), which require additional framing, buffering and processing logic. In other words, it is no wonder that a high-performance network application is anything but trivial to write.

Wouldn't it be nice if we could abstract some of the low-level details of different socket types, connection handling, framing, or even routing? This is exactly where the [**ZeroMQ**](http://www.zeromq.org/)**(ØMQ/ZMQ)** networking library comes in: *"it gives you sockets that carry whole messages across various transports like inproc, IPC, TCP, and multicast; you can connect sockets N-to-N with patterns like fanout, pubsub, task distribution, and request-reply"*. That's a lot buzzwords, so lets dissect some of these concepts in more detail.

**Message-Oriented vs. Streams & Datagrams**

ZeroMQ sockets provide a layer of abstraction on top of the traditional socket API, which allows it to hide much of the everyday boilerplate complexity we are forced to repeat in our applications. To begin, instead of being [stream (TCP)](http://en.wikipedia.org/wiki/Transmission_Control_Protocol), or [datagram (UDP)](http://en.wikipedia.org/wiki/User_Datagram_Protocol) oriented, **ZeroMQ communication is message-oriented**. This means that if a client socket sends a 150kb message, then the server socket will receive a complete, identical message on the other end without having to implement any explicit buffering or framing. Of course, we could still implement a streaming interface, but doing so would require an explicit application-level protocol.

# create zeromq request / reply socket pair

ctx = ZMQ::Context.new

req = ctx.socket ZMQ::REQ

rep = ctx.socket ZMQ::REP

# connect sockets: notice that reply can connect first even with no server!

rep.connect('tcp://127.0.0.1:5555')

req.bind('tcp://127.0.0.1:5555')

req.send ZMQ::Message.new('hello' \* (1024\*1024))

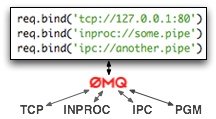
msg = ZMQ::Message.new

rep.recv(msg)

msg.copy\_out\_string.size # => 5242880

Switching from a streaming/datagram to a message-oriented model is seemingly a minor change, but one that carries a lot of implications. Because ZeroMQ will handle all of the buffering and framing for you, the client and server applications become an order of magnitude simpler, more secure, and much easier to write.

**Transport Agnostic Sockets**

ZeroMQ sockets are also transport agnostic: there is a single, unified API for sending and receiving messages across all protocols. By default, there is support for [in-process](http://api.zeromq.org/zmq_inproc.html), [IPC](http://api.zeromq.org/zmq_ipc.html), [multicast](http://api.zeromq.org/zmq_pgm.html), and [TCP](http://api.zeromq.org/zmq_tcp.html), and switching between all of them is as simple as changing the prefix on your connection string. This means we can start with IPC for fast local communication, and then switch to TCP at any point for distributed cases with minimal effort. As an added benefit, ZeroMQ handles all connection setup, teardown, and reconnect logic under the hood. That's about as simple as it gets.

**Routing & Topology Aware Sockets**

ZeroMQ sockets are routing and network topology aware. Since we don't have to explicitly manage the peer-to-peer connection state - all of that is abstracted by the library, as we saw above - **nothing stops a single ZeroMQ socket from binding to two distinct ports to listen to for inbound requests**, or in reverse, send data to two distinct sockets via a single API call. How does ZeroMQ know who to listen to or push data to? That depends on the type of the socket pair we pick for our application: *Request/Reply*, *Publish/Subscribe*, *Pipeline*, and *Pair* (alpha).

ctx = ZMQ::Context.new

# create publisher socket, and publish to two pipes!

pub = ctx.socket(ZMQ::PUB)

pub.bind('tcp://127.0.0.1:5000')

pub.bind('inproc://some.pipe')

# generate random message, ex: '1 9'

Thread.new { loop { pub.send [rand(2), rand(10)].join(' ') } }

# create a consumer, and listen for messages whose key is '1'

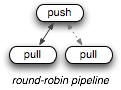
sub = ctx.socket(ZMQ::SUB)

sub.connect('inproc://some.pipe')

sub.setsockopt(ZMQ::SUBSCRIBE, '1')

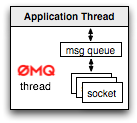
loop { p sub.recv } # => "1 9" ...

In the case of a [Publish/Subscribe](http://api.zeromq.org/2-1-1:zmq-socket#toc8) socket pair (unidirectional communication from publisher to subscribers), the publisher socket will replicate the message to all connected clients (local IPC clients, remote TCP listeners, etc). In the case of a [Request/Reply](http://api.zeromq.org/2-1-1:zmq-socket#toc3) socket pair (bi-directional communication: server, client), the messages will be automatically load balanced by the socket generating the request to one of the connected clients. Finally, a[Push/Pull](http://api.zeromq.org/2-1-1:zmq-socket#toc11) socket pair (pipeline: unidirectional, load-balanced) will allow you to simulate a staged message passing architecture with built-in load balancing.

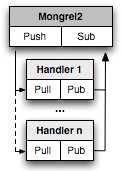
ZeroMQ allows us to encode the topology of our services directly via the socket API, without having to define and maintain a separate coordination layer of routers, load balancers, and message brokers. Of course, nothing stops us from using any of these tools in combination with ZeroMQ, but in many cases, the ZeroMQ route can yield [better performance](http://www.zeromq.org/area:results) and much simpler operational complexity.

**ZeroMQ under the hood**

By default, all communication in ZeroMQ is done in [asynchronous fashion](http://api.zeromq.org/zmq_socket.html#toc2). To enable this, anytime you create an application with ZeroMQ, you will have to explicitly declare the number of background I/O threads - in most cases, a single dedicated I/O thread will suffice. All of the thread logic is handled by the C++ core of the library itself, but it does mean that at very minimum, your application will have two scheduled threads.

This asynchronous processing model allows ZeroMQ to abstract all connection setup, teardown, reconnect logic, and also to minimize message delivery latency: no blocking means the messages can be dispatched, delivered and queued (sender or receiver side) in parallel to the regular processing done by your application. Of course, you can also control the queuing behavior of ZeroMQ sockets by setting an [allowed memory bound](http://api.zeromq.org/2-1-1:zmq-setsockopt#toc3) and even a[swap size](http://api.zeromq.org/2-1-1:zmq-setsockopt#toc4) for each socket. Hence, you can simulate the blocking API if desired, but asynchronous I/O is the default. Combined with zero copy semantics, optimized framing, and no locking data structures, the end result is a high performance and throughput oriented messaging middleware with a modern API.

**ZeroMQ in the Wild: Mongrel 2**

[Mongrel2](http://mongrel2.org/home) offers an interesting case-study of applying ZeroMQ to the world of web-servers: all inbound requests are routed by Mongrel2 via a "Push" socket which automatically load-balances the requests to connected handlers. The handlers, in turn, process the incoming requests (via Pull socket) and publish them to a "Pub" socket, to which the Mongrel2 server itself is subscribed to and is listening for its process ID (via a [topic filter](http://api.zeromq.org/2-1-1:zmq-setsockopt#toc7)).

Hence, the processing is not tied to a simple request-response cycle we are commonly used to, where a single backend has to handle the full request start to finish. Instead, we can setup[several processing stages](http://mongrel2.org/doc/tip/docs/manual/book.wiki#x1-660005.3) (via pipeline pattern), and emit our reply only after it is processed by all stages.

**Ambitious and worth exploring**

Needless to say, ZeroMQ is an ambitious project, and this short introduction only scratches the surface of the full feature set. The stated goal of ZeroMQ is to *"become part of the standard networking stack, and then the Linux kernel"*. Whether they will succeed, remains to be seen, but it is definitely a very promising and arguably a much needed layer of abstraction on top of the "traditional" BSD sockets. ZeroMQ makes writing high performance networking applications incredibly easy and fun.

The best way to get started with ZeroMQ is to work through some hands-on examples - the concepts are not new, but the ease with which you can compose them takes some getting use to. For Rubyists, Andrew Cholakian has put together a [great set of examples](http://github.com/andrewvc/learn-ruby-zeromq) to get you started (check out [dripdrop](http://github.com/andrewvc/dripdrop) as well), and for everyone else, head to the ZeroMQ site, grab [your language bindings](http://www.zeromq.org/bindings:_start) and dive into the code.