Building distributed applications with ZeroMQ, Part 1: Leverage ZeroMQ to build distributed applications

Implement an efficient and secure communications scheme

Daniel Robbins

October 16, 2017

In this three-part series, we are going to look at a very elegant solution to an extremely common problem on how to send and receive messages across the network efficiently and securely. In the age of cloud-based computing and 'Internet of Things' devices, this type of security is a critical need, so it makes sense to have something in your tool belt that will allow you to add this important capability to your applications.

Want to check out Daniel's other articles? Click here to read his past works.

It's been quite a while since I last wrote for developerWorks, and it's nice to be back. In this three-part series, we are going to look at a very elegant solution to an extremely common problem on how to send and receive messages across the network efficiently and securely. In the age of cloud-based computing and 'Internet of Things' devices, this type of security is a critical need, so it makes sense to have something in your tool belt that will allow you to add this important capability to your applications.

We live in an era where applications are often distributed—a single distributed application might have parts that are running in a cloud service and other parts that are running in a physical data center. Some devices can also be 'out in the field,' at random client sites.

The common denominator among all these application components is often the public Internet. It is extremely useful to be able to implement an efficient and secure communications scheme that works over the Internet without requiring any additional infrastructure, such as virtual private networks that can significantly increase cost and complexity.

While there are a lot of different approaches to solving this problem of distributed communication, we are going to look at a particularly elegant approach. This approach will allow you to embed an advanced, secure, Internet-friendly, and message-based communication within application components—they essentially become 'super charged' and are able to solve this thorny communication problem without any additional help. This solution can be used from nearly any programming language and will provide you with modern functions, like advanced encryption and

support for sophisticated messaging schemes. This technology is well suited for everything from the most petite Raspberry Pi-embedded device, to the cloud, to beefy physical servers housed in a data center.

Enter ZeroMQ

One common solution to facilitating communication between disparate application components is to use a message queue. Most traditional message queues act as 'hubs' to facilitate the exchange of messages over the network between distributed software components. Applications send messages to a central queue by using some kind of API and are used for various purposes.

> *“ ZeroMQ is actually not a message queue. It is a C library that is kind of like a message queue construction kit, which allows for advanced messaging functionality to be integrated into practically any application. ”*

First, a bit about ZeroMQ. ZeroMQ has quite an interesting history. ZeroMQ was created by iMatix, the original designers of the AQMP message queuing protocol. In 2010, Pieter Hintjens, then CEO of iMatix, announced that iMatix was formally leaving the AQMP working group and instead would be focusing on ZeroMQ, which they considered to be a better approach to solving the problem of distributed messaging. Real-world experience showed that ZeroMQ is very efficient and capable. By creating the ability for application components themselves to be first-class members of distributed messaging solutions, the solutions can be leaner and more tailored to their specific problem domain.

More comfortable with Apache Kafka? Check out the Message Hub here.

Once you are familiar with the particulars of ZeroMQ, all of these advanced functions are possible with minimal code. To me, the beauty of ZeroMQ is the power that it gives you. It allows you, the programmer, to implement a messaging scheme that makes sense for your particular application. You can simply implement one with minimal code that does everything you could possibly need and want, with an incredibly small footprint—no middleware required!

Get caught up on Python programming here.

ZeroMQ also has bindings for a ton (really, a lot!) of different languages, including Python, which we will use in this article series due to its approachability. The goal of this article series is to leave you with working production-quality code to solve the common problem of distributed messaging, along with the knowledge and confidence to adapt it to your needs. The code in this article will be easily adaptable to your language of choice, whether that happens to be C, C++, Haskell, Java™, PHP, Ada, or something else.

Are there any downsides with ZeroMQ? Perhaps it requires a bit more reflection on the overall architecture of your distributed application. And it becomes your responsibility to define the logic for how your application communicates—while not necessarily difficult, it does require some additional work. However, if you crave this level of control and flexibility, then this extra work isn't a negative. It ultimately allows you to tailor the communications exchanges to your specific needs

and tightly integrate them into your application's logic. This gives you the ability to get exactly the kind of results you want from your distributed communications scheme.

ZeroMQ provides an API for distributed messaging that, to the programmer, looks a lot like typical raw TCP sockets. But unlike a single TCP socket, which can only facilitate direct communication between two applications, ZeroMQ's API transparently handles an arbitrary number of peer connections, abstracting away much of the complexity of distributed messaging. So, ZeroMQ's API will likely look familiar at first glance. And I will guide you through the conceptual hurdles.

For the purposes of this series, we will use PyZMQ 15 or later, which are Python bindings for ZeroMQ. The code I present in this article is compatible with Python 3 and can also run under Python 2 with slight modifications. Along the way, I will share my personal best practices, approaches, code snippets, and conceptual guidance to hopefully propel you rapidly towards leveraging ZeroMQ within your own projects.

But you say you're a sysadmin and not a programmer? That's okay—this article series still offers something for all you sysadmins. Not only will I give you a good feel for how ZeroMQ works internally, but check out the other articles in this series to see where I will provide you with a turnkey ZeroMQ-based framework that you can leverage and extend for your own needs. With minimal Python experience, you'll be able to write your own code to monitor important metrics in a data center and send them securely over the network—all without having to use ZeroMQ directly. You will need to write very, very few lines of your own Python code.

Messages and frames

At the heart of ZeroMQ is *the message*. After all, this is a message-based protocol. We will be using multi-part messages. Each message part is called a 'frame.' Each frame can hold any kind of data you want, and each frame records the length of its data so ZeroMQ can figure out where the frame begins and ends. ZeroMQ guarantees that it will deliver your message in its entirety, or will fail to deliver it at all. So you can rest assured that your code does not need to worry about receiving or sending partial messages.

One very useful convention is to make the first frame of your message a short ID string that will be used by your client and server software to determine the type of message it is handling. This way, your software can easily determine the type of message it is handling and it will know how to parse successive frames.

Another useful convention is to define a multi-purpose message format that can be used for various purposes. While it is possible to define a unique ZeroMQ message type for each kind of operation, it can be more convenient to have as few message types as possible—this will reduce the amount of code you need to write to parse each message type.

Let's look at some sample Python code that defines a class for helping us manage ZeroMQ messages:

```
class MultiPartMessage(object):

 header = None

 @classmethod
 def recv(cls, socket):
  "Reads key-value message from socket, returns new instance."
  return cls.from_msg(socket.recv_multipart())

 @property
 def msg(self):
  return [ self.header ]

 def send(self, socket, identity=None):
  "Send message to socket"
  msg = self.msg
  if identity:
   msg = [ identity ] + msg
  socket.send_multipart(msg)
```

From the sample code above, the class, `MultiPartMessage`, is designed to be an abstract class; that is, it is designed to be subclassed and not used directly. Now, let's create a very simple example of how to use `MultiPartMessage`. Let's create a message that has only one frame, the string header itself:

```
class HelloMessage(MultiPartMessage):

 header = b"HELLO"
```

The above code defines a "HELLO" message that contains no data. It's legitimate, but potentially not very useful. But it's the simplest working example of a message. Let's look at how we might use this in ZeroMQ code:

```
my_msg = HelloMessage()
my_msg.send(my_zeromq_socket)
```

Above, we pass a valid ZeroMQ socket to the send method, and the message will be sent through this socket. As you can see, the API is pretty simple so far. Now let's see how to define a more complex message:

```
class FileMessage(MultiPartMessage):

 header = b"FILE"

 def __init__(self, filename, contents):
  self.filename = filename
  self.contents = contents

 @property
 def msg(self):
  # returns list of all message frames as a byte-string:
  return [ self.header, self.filename.encode("utf-8"), self.contents ]
```

From the above sample code, we have a type of message that is called a `FileMessage`. All FileMessages will have a header (first frame) containing "FILE." The second frame will contain the file's name, and the third frame will contain the actual binary data. Notice that we use the

`.encode()` method to encode the file name in UTF-8 format—this will return a byte stream, which is important to note. Remember that our message is going to be sent over the wire, which means that ZeroMQ will convert our data to a byte stream. Above, we do not encode the contents of the file, since we assume that it is already a byte string. Here's an example of how we might use this class:

```
filename = "myfile.tar.gz"

with open(filename, "rb") as myfile:

 # read contents of file into a byte-string, rather than a python
 # string, thanks to 'rb' above. This allows ZeroMQ to send it over
 # the wire as a series of bytes.

 contents = file.read()

 my_msg = FileMessage(filename, contents)
 my_msg.send(my_zeromq_socket)
```

This encoding issue is an important caveat. Your clients, servers, and middleware will need to agree upon the encodings for the various frames of your message. As I did in the FileMessage class, I recommend that you use a single shared class on both the client and server to centralize the encoding of all data so that encoding is handled by a single unified code-base. That means that once the message format is designed, your code can use it without paying much attention to the format.

This raises the question of where is the code that converts a received ZeroMQ message to a FileMessage class? I typically use a `from_msg()` class method, as seen in the following snippet.

```
 @classmethod
 def from_msg(cls, msg):
  "Construct a FileMessage from a pyzmq message"
  if len(msg) != 3 or msg[0] != cls.header:
   #invalid
   return None
  return cls(msg[1].decode("utf-8"), self.contents)

my_msg = FileMessage.from_msg(zeromq_msg)
```

If your server code is receiving multiple kinds of messages, you'll want to first inspect the first frame of the message to determine its type, and then pass it to the proper constructor to 'dehydrate' it and create a live Python object. This is easy because typically you will receive your message as a list of byte-string frames:

```
def handle_message(self, msg_parts)
 if msg[0] == b"FILE":
  orig_msg = FileMessage.from_msg(msg_parts)
 elif msg[0] == b"HELLO":
  ...
 else:
  print("ERROR !! I don't recognize this message!")
```

Now let's take a look at the example code. We have a client that's defined in `app_client.py` and a server that's defined in `app_server.py`. Both should work with PyZMQ 15 and above, so

using your distribution's package manager to install `pyzmq` should suffice to bring in all necessary dependencies.

Our client and server are currently designed to do one simple thing, which is saying "HELLO" to each other. Once per second, they will send a "HELLO" message across the wire. It is important to note that we are using a ROUTER-DEALER pattern, which is a ZeroMQ pattern that allows full asynchronous communications. This means that, as in our example, our client and server are free to send a message to each other whenever they want, rather than being limited to only being able to respond to an incoming message, for example. Other simpler patterns are not as flexible and enforce a lock-step ping-pong communications pattern. In our sample code, we skip directly to the good stuff that we'd want to use in production.

Grab the code now

Let's do a walkthrough of the `app_client.py` code. The AppClient() class has a constructor that tries to connect to localhost port 5556, the address where `app_server.py` will be listening. It also creates a periodic task that runs once per second, and tells our client ZMQStream that it should call our `on_recv()` method when a message is received.

The `start()` method actually starts our asynchronous loop and our periodic task. Every second, we try to send a "HELLO" message to the server. We also see if we received a "HELLO" message from the server in the last 5 seconds. If we haven't, we print a message that indicates the server is unresponsive, but we keep trying to send "HELLO" to the server. The client will run forever until interrupted.

Now, it is entirely possible to extend the client to do lots of interesting things. Thanks to the periodic task, you have the ability to perform various things, such as read from a database or collect performance metrics. As long as the tasks that you perform won't block for extended periods of time, the code will run reliably. If you need to do something like retrieve data from a remote server over the Internet as part of your client, this part of the code should be done asynchronously. PyZMQ's async implementation is borrowed from tornado and is compatible with it, so you can use tornado conventions to add other asychronous routines to the client.

The example code: Server

Now, let's take a look at the server code, `app_server.py`. The `AppServer()` constructor will listen on localhost port 5556 for incoming ZeroMQ connections. It can actually handle any number of clients simultaneously so that a single connected client will not prevent others from communicating with our server. Similar to the client, our server also registers a periodic task that will run once per second, and an `on_recv()` method that will be called when it receives a message.

If you take a look at `on_recv()` and `periodictask()`, you'll note that our server first 'learns' of a client connection when it receives a message from it. The first frame of the received message contains the 'identity' of the client, which is a byte string (generated by the ZeroMQ library) that uniquely identifies the client connection. You can see that `on_recv()` extracts this identity and then stores it within `self.client_identities`, a dictionary with a timestamp of when this message was received. This is the behavior of the router in ZeroMQ—it tracks each incoming connection and

generates an identity for the connection to be used by our code. Each message that is received from this connection is prepended with the connection's unique identity as its first frame.

In `periodictask()`, which runs once per second, our server will iterate through its list of known clients, and if it heard from this client in the last 10 seconds, it will attempt to send a HELLO message to it. Note that it uses the `client_identity` to direct the message to a specific client. If it did not hear from the client in the last 10 seconds, it will be considered a 'stale' client and will be removed from the list of active client connections.

Reflections

Play around with the client and server. Run them both on the same machine. Here are some things to attempt to see how the client and server respond. Perform these tests and see whether you can understand why the code behaves how it does.

1. Run a single client and server. Do the messages output to the console make sense?
2. Run a single client and server and then kill the client. Wait 20 seconds and then start a new client.
3. Start the client, wait 20 seconds, and then start the server. Notice how the server receives many HELLO messages immediately. What do you think happened?
4. Start a client and server. Then kill the server. Now, restart the server.
5. Start two or more clients and a single server.

One concept that is important to point out is that neither the client nor the server code deal directly with a single client/server 'connection' within the code. The client connects, and attempts to send messages to the server. It doesn't know whether the connection to the server is active or not. Behind the scenes, the ZeroMQ library is trying to connect to the server on behalf of the client. When it connects, any messages that the client attempted to send to the server will be delivered.

Likewise, the server has no immediate concept of an individual server-client connection. It becomes aware of a client when it receives a message from a client with an identity that it has never seen before. It also does not know when an individual client disconnects. Instead, it assumes that if it didn't receive a message from the client in the last 10 seconds, that the client connection is no longer active.

This level of abstraction is intentional. By allowing the ZeroMQ library to manage individual connections, our code can focus on the communications protocol instead of spending a lot of effort in managing resources that are related to the network communication. This allows our code to focus on the messages that are being exchanged and remain blissfully unaware of the complex networking details that are happening within the ZeroMQ library itself.

That's it for now. Try playing with the code and see if you can make it do some interesting things. Next time, we'll take our currently unencrypted communications channels and make them secure against eavesdropping and tampering. See you then!

Related topics

- Don't forget to grab the code from this article
- Distributed applications
- Python
- Send and receive messages
- Tornado
- ZeroMQ
- Create Python apps with IBM Watson and IBM Bluemix
- MQ on Cloud
- Create autoscaling actions that respond to message streams