Introduction to ZeroMQ

Luis Flores

2024-06-11

Introduction to ZeroMQ

> Author: Luis Flores
>
>

In this document we will look at how we can use ZeroMQ to send and receive messages. ZeroMQ provides us with an API that allows us to send a message from one program to another.

If you are building microservices, your microservice would typically be the server, and the programs that interact with this server would be considered your client. For example, if you build an application that needs to receive a random number it would be the client since it would need to make the call to a server to get that value. That microservice (server) would process the request and send the data back to your program (client).

The typical ZeroMQ process typically involves having a "server" and a "client".

- The "server" in this case can be thought of as **static** largely staying unmodified through the program lifecycle.
- A "server" typically binds (connects).
- The "client" is considered **dynamic**.
- A "client" which reacts depending on the specific task of either sending or receiving a message.

For now, we will specifically be looking at two main patterns: **Request-Reply** and **Publish- Subscribe**.

Key Terms

Before we begin looking at how to use ZeroMQ in our code, let's first define and get familiar with some terms that are mentioned throughout this document.

Server: The program that listens for a request from a program.

Client: The program that requests information from another program.

Port: Number to identify where connection will be directed.

localhost: This a reference to your computer

IP Address: A unique address that identifies your computer.

Socket: Locations where data is sent and received.

Request-Reply

Request-Reply: This works by having a "server" and a "client". The client sends a message and the server does something based on the request. The server will receive a message and send an appropriate response. The client will then receive the response from the server and continue doing its work. Here is the process laid out: - Client sends a message to the server via zmq_send() - Server receives the response with zmq_recv() - Server can now send back a response with zmq_send() - Client receives the response from the server with zmq_recv() - This process can be done endlessly using a loop or it can be used just once, so you can modify it depending on your use case.

> Note: The process must start with the client, as the server cannot initiate the first send.

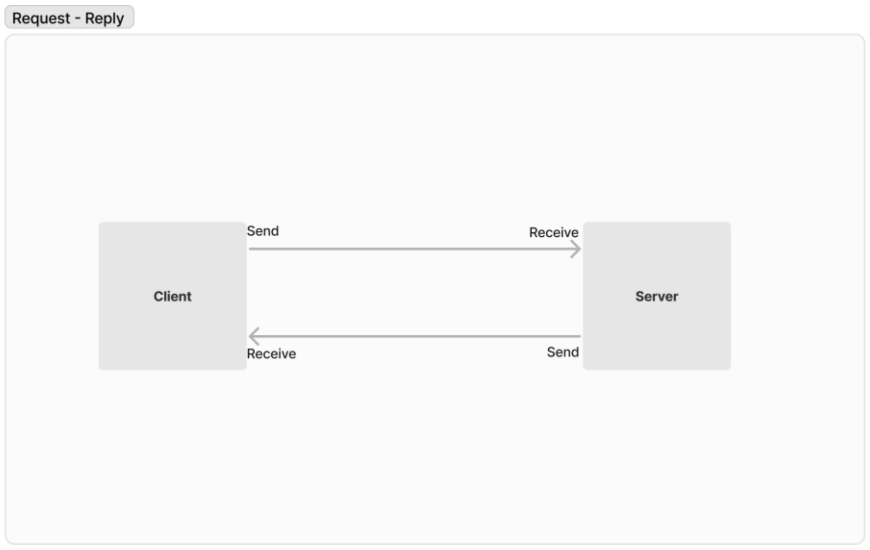

Figure 1: Request Reply Pattern

Installing ZeroMQ (Python)

To get started we will need to install ZeroMQ. Using a terminal, we can install the package using the following command:

```
pip install pyzmq
```

Note: If you are having problems make sure you have pip installed on your machine. Visit the installation guide found here for more details about installation.

A simple Request-Reply

> The REQ-REP pattern is blocking! The REQ (request) will send a message and wait for a response, the REP (reply) will accept a message and stops receiving messages until it replies.

The server

Once you have ZeroMQ installed we can begin working on our program. First, we will create a new python file and title it server.py

This file will contain the static portion. It will bind to a specified port and wait for a message from the client and perform some sort of work before sending a message.

First let's get started by including our import statements:

```
#   This code connects (binds) to port 5555 on your local
#   machine, it then waits for a message from the client
#   when the server gets a message it performs some action
#   and sends a message back to the client.
#
#   This code was largely based on the example found on
#   zguide  titled  hwserver:  https://zguide.zeromq.org/docs/chapter1/
#   (c) 2010-2012 Pieter Hintjens

import time
import zmq # For ZeroMQ
import random
```

Next, we need to begin working with ZeroMQ. We will create one context for our file. We will also declare which ZeroMQ socket we will be using.

```
# @Context(): sets up the environment so that we are able to begin
# creating sockets. This is required for every file using
# ZeroMQ and should only be declared once at the start.
context = zmq.Context()
```

```
# @socket(socket_type): This is the type of socket we
# will be working with. In this case REP is the reply socket.
socket = context.socket(zmq.REP)

# @bind(addr): This is the address string. This is
# where the socket will listen on the network port.
# The format for this is: protocol://interface:port
socket.bind("tcp://*:5555")
```

Now that we have everything setup, we now need to make the program do some work!

First, we will create a loop and listen until a message is received from the client. Once we get a message, we can then perform a task and send the client back a response. If the client sends us a message telling the server to quit, we will exit the loop and end our context.

```
# This will create an infinite loop that will wait for
# a message from the the client.
# Once a message is recevied it will generate a
# random number and send it back to the client.
while True:
    # Message from the client.
    # @recv(flags=0, copy: bool = True, track: bool = False): will
    # receive a message from the client.
    # This will be blank since we will just wait until the message arrives.
    message = socket.recv()

    # We will decode the message so that we don't get a 'b' in front of text
    # ZeroMQ defaults to UTF-8 encoding when nothing is specified.
    print(f"Received request from the client: {message.decode()}")

    if len(message) > 0:
        if message.decode() == 'Q': # Client asked server to quit
            break

        # This is where we can perform 'work' or
        # tasks that we want our program to do.
        # Generate a random number.
        num = random.randint(1, 5)

        # Make the program sleep for X seconds.
        time.sleep(3)

        # NOTE: If you are sending a int back you
        # will need to convert it to a string since
        # zeroMQ communicates with raw bytes.
        # Alternatively, you can use a struct.
        myNum = str(num)

        # Send reply back to client
        # @send_string(): sends a string
        socket.send_string(myNum)

# Make a clean exit.
context.destroy()
```

The client

Once we have that created, we can begin working the client. The client will connect to the server and send it a message. The server will then respond with a message which the client will open with a receive.

We will create a new python file and title it client.py.

First, we will start by importing ZeroMQ!

```
#   Client for ZeroMQ request-reply program. This will connect to
#   a server via a request socket. Once connected to the localhost
#   sends a string message and waits for reply from the server.
#
#   This code was heavily based on the example found on
#   zguide titled hwclient: https://zguide.zeromq.org/docs/chapter1/

import zmq # For ZeroMQ
```

Next, we will create a single context for our file. We will include a print statement so that we can see what our program is doing. Then we will create a reply socket to work with. Last, we will have this socket connect.

```
# @Context(): sets up the environment so that we are able to begin
# creating sockets.
context = zmq.Context()

# Connect to the server to send a message.
print("Client attempting to connect to server...")

# @socket(socket_type): This is the type of socket we
# will be working with. In this case REQ is the request socket.
socket = context.socket(zmq.REQ)

# @connect(addr): This will connect to a remote socket
# The format for this is: protocol://interface:port
socket.connect("tcp://localhost:5555")
```

Once we have gone through all the setup we can now have our client send a request to the server. We will ask the server to generate a random value for us. We will then wait for a response, once we get our message, we will ask the server to stop running by sending a Q representing quit since we will no longer be using it.

```
# Request a value from the server. Sending a
# user specified string.
print(f"Sending a request...")

#@send_string will send the user specified string.
socket.send_string("Generate a value")

# Get the reply.
#recv(flags=0, copy: bool = True, track: bool = False): will
# receive a message from the client.
# Parameter will be blank for simplicity.
message = socket.recv()

# Print our message!
print(f"Server sent back: {message.decode()}")

#End server
socket.send_string("Q") # (Q)uit will ask server to stop.
```

Bidirectional Communication: Two-way Requests and Replies

In the last example we saw how to use the request reply pattern to send and receive messages. This process is straightforward with a client and server, but what if we wanted to initiate the request from either end? The code won't be able to accomplish this because a REP client is unable to initiate a request.

We need to use an asynchronous pattern if we want to send multiple messages without receiving a reply. Earlier we saw the REQ-REP was blocking. This meant we could not send multiple messages!

To accomplish our goal of initiating a request from the client or server we need to use a DEALER type socket. For now, we will use a DEALER-DEALER pattern. This pattern does not scale well. In our use case, we will only be talking to one other peer so there is no problem.

Advantages for DEALER-DEALER * Asynchronous this means its non-blocking! * In this pattern the socket can take the role of a reply or request socket

Disadvantages for DEALER-DEALER * Does not scale well - a lot of messages can cause it to become blocking * Difficult to handle messages if not set up correctly.

In this example both programs will be a client and server!

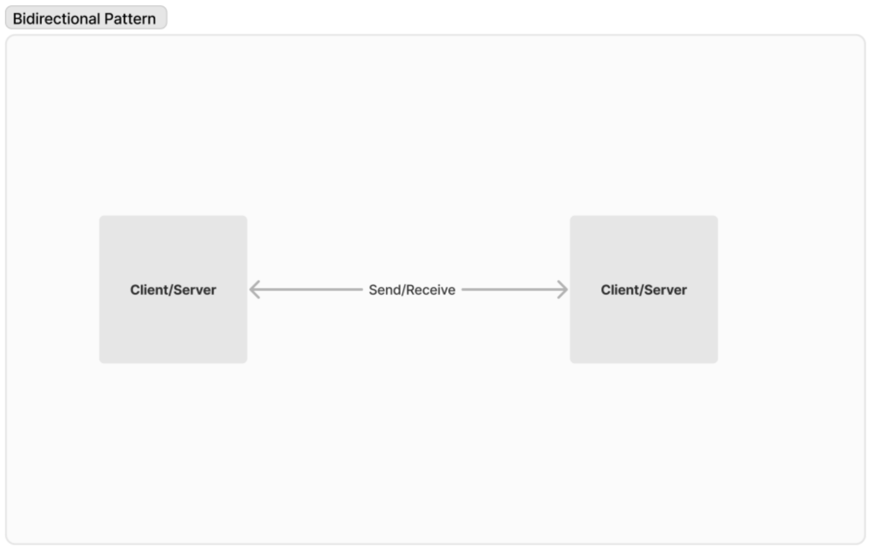

Figure 2: Bidirectional Pattern

This is an over simplified version of what is happening behind the scenes. However, it does let us see that either program can initiate a request.

Let's begin by creating a python file titled peer1.py . Through this we will be able to both initiate and receive messages from either side. We can ask for a random number and also receive the value from the same program.

To start we will need to include our import statements

```
#    Client/Server for ZeroMQ. This will bind to a
#    server and client, allowing it to start sending #
     and receiving data.
#    Server binds to tcp://*:5555
#    Client connects to tcp://localhost:5556

import zmq # For ZeroMQ
import random
import time
```

Next, we need to initiate our context for this file, remember you only need to do this once per file! We will also create two sockets, a client and server so that we either initiate or listen for a message.

```
# @Context(): sets up the environment so that we are able to begin
# creating sockets.
context = zmq.Context()

# @socket(socket_type): This is the type of socket we
# will be working with. In this case DEALER is the request socket.
# One will be a server type and the other a client.
socket_server = context.socket(zmq.DEALER)
socket_client  = context.socket(zmq.DEALER)


# @bind(addr): This is the address string. This is
# where the socket will listen on the network port.
# The format for this is: protocol://interface:port
socket_server.bind("tcp://*:5555")

# @connect(addr): This will connect to a remote socket
# The format for this is: protocol://interface:port
socket_client.connect("tcp://localhost:5556")
```

Next, we will do something a little different. Instead of directly writing the code to have our program do some work, will create something called a zmq.Poller() to help us handle receiving messages without causing our program to block.

```
# This will help us listen for specific events like messages
poller = zmq.Poller()


# Add that poller to monitor our server socket.
poller.register(socket_server,  zmq.POLLIN)
```

Once we have a poller, we can now give our program a task. We will assign our program to send a message and then wait for a message to appear. When we get a message, we can generate a number and end our program.

```
# Send request.
socket_client.send_string("Random")
print("Sent a request...")

# Loop waiting for a message. When a message arrives
# we will create a random number and wait 3 seconds to
# send it to a client.
while True:
    # Handles inputs occurring for the socket.
    socks = dict(poller.poll())
    # Checks if something was received.
    if socket_server in socks and socks[socket_server] == zmq.POLLIN:
        print("Received request")
        # pseudorandom number
        num = random.randint(1, 5)
        # sleep (wait to send)
        time.sleep(3)
        # Send the value as a string.
        socket_server.send_string(str(num))
        break

# Receive message from the client.
message   =   socket_client.recv()

# Print our message!
print(f"Server sent back: {message.decode()}")

# Exit cleanly - terminates context.
context.destroy()
```

Peer 2 will look almost identical as it serves the same function. For peer2.py We will need to modify the following two lines:

```
# Modified to 5556
socket_server.bind("tcp://*:5556")

# Modified to 5555
socket_client.connect("tcp://localhost:5555")
```

The Publish-Subscribe Pattern

The next pattern we will look at will be **publish-subscribe**. This pattern typically works by having a publisher and either one or more subscribers. A publisher will send out messages for specific topics. A publisher will then subscribe to one of these topics and receive the message contained within.

The publish-subscribe pattern works similarly to the reply-request in the way the sockets connect and bind. For example, a publish socket will **bind** just like a reply socket. On the other hand, a subscribe socket and request socket will both **connect**.

For now, we will examine a program that has one publisher and one subscriber. For this program the **publisher** will have a list of topics (the days of the week) and produce a message for each day (a joke). The subscriber will choose a topic (day of the week) and then receives the message (the joke for that day). Here is the process laid out:

- Publisher binds to an address (5555)
- Subscriber connects to a port (5555)
- Publisher produces a message for each topic and sends it out
- Subscriber chooses a topic via socket option

- Subscriber receives the message from the selected topic when the publisher sends it out.

The diagram below demonstrates how a publisher sends out a message to a subscriber for a specific topic. The grey dashed boxes show that there could be more than one subscriber for one publisher, each subscribing to the same or different topic.

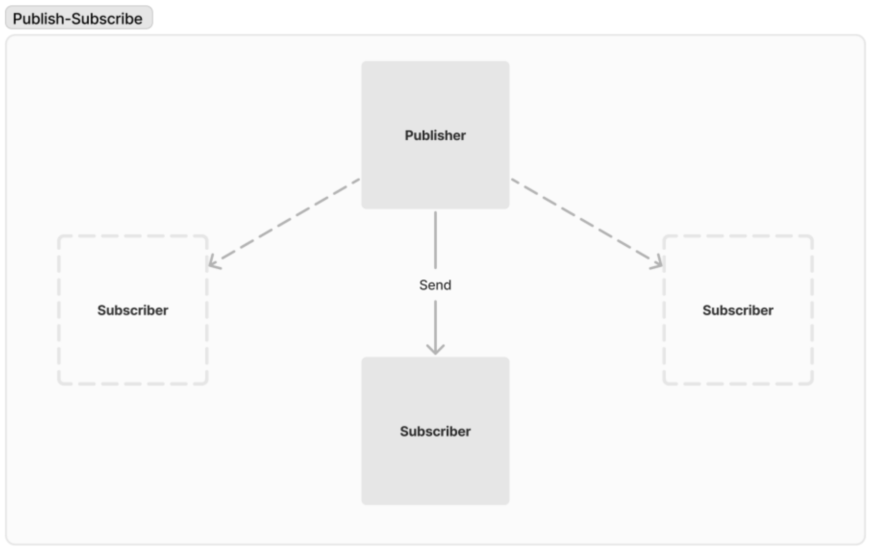

Figure 3: Publish Subscribe Diagram

The Publish Socket

Next, let's take a look at how the code for this process looks. First, we will create a file pub.py this serves as the publisher.

First, we need to set up the file with the following code. This is pretty similar to previous code in this document as we are just creating our ZeroMQ context and socket to work with.

```
#   @author: Luis Flores
#   Subscriber for ZeroMQ publish-subscribe program. This will connect to
#     a server via a publisher socket. Once connected to the localhost
#     it will wait to receive a message for a specific topic.
#     It uses pyjokes installable via pip: pip install pyjokes
#
# This code was heavily based on the example found on
#   learning-0mq-with-pyzmq.readthedocs.io/en/latest/pyzmq/patterns/p
import zmq # For ZeroMQ
import random
import time
import pyjokes # generates the jokes

# @Context(): sets up the environment so that we are able to begin
# creating sockets.
context = zmq.Context()

# We need a socket to work with, we will use a Publisher (PUB) socket
# for this example
socket = context.socket(zmq.PUB)

# @bind(addr): This is the address string. This is
# where the socket will listen on the network port.
socket.bind("tcp://*:5555")
```

> Note you can install pyjokes via pip with the following command:
> pip install pyjokes

Next, we will need a list of topics. For this example, we will use the 7 days of the week.

```
# Each item is a day of the week.
categories = ["Monday", "Tuesday", "Wednesday",
              "Thursday", "Friday", "Saturday", "Sunday"]
```

We then need to create a while loop to have our publisher produce messages indefinitely. Inside this loop we will choose one of the seven days of the week as our topic. We then will generate a message in the form of a joke to go along with this topic. Once we have both we will then send that string out and a subscriber should pick it up if it is one of the topics they have subscribed to.

```
# Creates an infinite loop and generates the message to be sent for
#  subscribers. The topic is chosen from the seven days of the week,
#  and a message is the joke returned from pyjokes.
while True:
    topic = categories[random.randrange(7)]
    message = pyjokes.get_joke()
    print(f"A joke has been generated for {topic}")
    print(f"Sent: {message}")
    socket.send_string(f"{topic}, {message}")
    time.sleep(3)
```

The Subscribe Socket

We now need to focus on our subscribe socket. To get started we will create a file and name it sub.py.

Once again, we need to set up our file. In a similar manner as before we do the following: Import zmq and create the ZMQ context to work with. Once the context is created, we can then create our socket. We will now use a subscriber socket (zmq.SUB). Instead of binding like our subscriber, our publisher will now connect.

```
#    @author: Luis Flores, Spring 2024
#      Subscriber for ZeroMQ publish-subscribe program. This will connect to
#      a server via a subscribe socket. Once connected to the localhost
#      it will wait to receive a message for a specific topic.
#
#      This code was heavily based on the example found on
#    learning-0mq-with-pyzmq.readthedocs.io/en/latest/pyzmq/patterns/pubsub.html
import zmq # For ZeroMQ

# @Context(): sets up the enviornment so that we are able to begin
# creating sockets.
context = zmq.Context()

# Display connecting to server message
print("Client attempting to connect to server...")

# @socket(socket_type): This is the type of socket we
# will be working with. In this case SUB is the subscribe socket.
socket = context.socket(zmq.SUB)

# @connect(addr): This will connect to a remote socket
socket.connect("tcp://localhost:5555")
```

Now we need to select a topic for our subscription. Since we had days of the week, we need to choose a day, for now we will choose Friday. This means each Friday we will get a random joke.

```
# Subscribe to a topic (day of the week)
topic = "Friday"
```

All that is left to do is wait for the publisher to send a message for Friday (our topic). Once we receive the message, we can display it!

```
# Have the socket look for the messages
# regarding the chosen topic.
socket.setsockopt_string(zmq.SUBSCRIBE,   topic)

while True:
    # Get the reply from the subscriber
    message = socket.recv()

    # View our message
    print(message)
```

The result

Here is small sample of what the **publisher** sends out:

```
A joke has been generated for Sunday
Sent: .NET was named .NET so that it wouldn't show up in a Unix
directory listing.

A joke has been generated for Friday
Sent: What's the object-oriented way to become wealthy? Inheritance.

A joke has been generated for Tuesday
Sent: The best thing about a Boolean is even if you are wrong, you
are only off by a bit.
```

Since our subscriber has been only subscribed to Friday messages the only message it sees is the Friday one.

```
'Friday, What's the object-oriented way to become wealthy? Inheritance.'
```

Conclusion

Thats it! We have gone through two main patterns in ZeroMQ and have seen how to create programs that are like servers and clients. This document has only scratched the surface of what's possible with ZeroMQ but, hopefully it has given you a good starting point. If you would like to read more on ZeroMQ you can find more detailed explanations here: ØMQ - The Guide or here ZeroMQ.