How To Work with the ZeroMQ Messaging Library

Authored by: ***ASPHostServer Administrator*** *[asphostserver@gmail.com]*
Saved From: *http://faq.asphosthelpdesk.com/article.php?id=270*

ZeroMQ

ZeroMQ is a library used to implement messaging and communication systems between applications and processes - fast and asynchronously.

If you have past experience with other application messaging solutions such as RabbitMQ, it might come a little bit challenging to understand the exact position of ZeroMQ.

When compared to some much larger projects, which offer all necessary parts of enterprise messaging, ZeroMQ remains as just a lightweight and fast tool to craft your own.

This Article

Although technically not a framework, given its functionality and the key position it has for the tasks it solves, you can consider ZeroMQ to be the backbone for implementing the actual communication layer of your application.

In this article, we aim to offer you some examples to inspire you with all the things *you* can do.

Note: We will be working with the Python language and its classic interpreter (Python C interpreter) in our examples. After installing the necessary language bindings, you should be able to simply translate the code and use your favorite instead without any issues.

Programming with ZeroMQ

ZeroMQ as a library works through sockets by following certain network communication patterns. It is designed to work asynchronously, and that's where the MQ suffix to its name comes - from thread queuing messages before sending them.

ZeroMQ Socket Types

ZeroMQ differs in the way its sockets work. Unlike the synchronous way the regular sockets work, ZeroMQ's socket implementation "present an abstraction of an asynchronous message queue".

The way these sockets work depend on the type of socket chosen. And flow of messages being sent depend on the chosen patterns, of which there are four:

- **Request/Reply Pattern:**Used for sending a request and receiving subsequent replies for each one sent.

- **Publish/Subscribe Pattern:**Used for distributing data from a single process (e.g. publisher) to multiple recipients (e.g. subscribers).

- **Pipeline Pattern:**Used for distributing data to connected nodes.

- **Exclusive Pair Pattern:**Used for connecting two peers together, forming a pair.

ZeroMQ Transport Types

ZeroMQ offers four different types of transport for communication. These are:

- **In-Process (INPROC):** Local (in-process) communication transport.

- **Inter-Process (IPC):** Local (inter-process) communication transport.

- **TCP:** Unicast communication transport using TCP.

- **PGM:** Multicast communication transport using PGM.

Structuring ZeroMQ Applications

ZeroMQ works differently than typical and traditional communication set ups. It can have either side of the link (i.e. either the server or the client) bind and wait for connections. Unlike standard sockets, ZeroMQ works by the notion of knowing that a connection might occur and hence, can wait for it perfectly well.

Client - Server Structure

For structuring your client and server code, it would be for the best to decide and elect one that is more stable as the *binding* side and the other(s) as the *connecting*.

Example:

```
Server Application                          Client Application
--------------------[ < .. < .. < .. < .. ......................
Bound -> Port:8080                         Connects <- Port:8080
```


PDF Generated by PHPKB Knowledge Base Script

Client - Proxy - Server Structure

To solve the problems caused by both ends of the communication being in a dynamic (hence unstable) state, ZeroMQ provides networking devices (i.e. utensils out of the box). These devices connect to two different ports and route the connections across.

- **Streamer:** A streamer device for pipelined parallel communications.
- **Forwarder:** A forwarding device for pub/sub communications.
- **Queue:** A forwarding device for request/reply communications.

Example:

```
Server App.            Device | Forward          Client App.
  ............ > .. > . ]-----------------[ < .. < .. .........
    Connects               2 Port Binding            Connects
```

Programming Examples

Using our knowledge from the past section, we will now begin utilizing them to create simple applications.

Note: Below examples usually consist of applications running simultaneously. For example, for a client/server setup to work, you will need to have both the client and the server application running together. One of the ways to do this is by using the tool Linux Screen. To install screen on a CentOS system, remember that you can simply run:`yum install -y screen`.

Simple Messaging Using Request/Reply Pattern

In terms of communicating between applications, the request/reply pattern probably forms the absolute classic and gives us a good chance to start with the fundamental basics of ZeroMQ.

Use-cases:

-

For simple communications between a server and client(s).

-

Checking information and requesting updates.

-

Sending*checks*and updates to the server.

-

Echo or ping/pong implementations.

Socket type(s) used:

- zmq.REP
- zmq.REQ

Server Example: server.py

Create a "server.py" using **nano** (`nano server.py`) and paste the below self-explanatory contents.

```
import zmq
# ZeroMQ Context
context = zmq.Context()
# Define the socket using the "Context"
sock = context.socket(zmq.REP)
sock.bind("tcp://127.0.0.1:5678")
# Run a simple "Echo" server
while True:
    message = sock.recv()
    sock.send("Echo: " + message)
    print "Echo: " + message
```

When you are done editing, save and e xit by pressing CTRL+X followed with Y.

Client Example: client.py

Create a "client.py" using **nano** (`nano client.py`) and paste the below contents.

```
import zmq
import sys
# ZeroMQ Context
context = zmq.Context()
# Define the socket using the "Context"
sock = context.socket(zmq.REQ)
sock.connect("tcp://127.0.0.1:5678")
# Send a "message" using the socket
sock.send(" ".join(sys.argv[1:]))
print sock.recv()
```

When you are done editing, save and exit by pressing CTRL+X followed with Y.

Note: When working with ZeroMQ library, remember that each thread used to send a message (i.e. `.send(..)`) expects a `.recv(..)`to follow. Failing to implement the pair will cause exceptions.

Usage

Our`server.py`is set to work as an "echoing" application. Whatever we choose to send to it, it will send it back (e.g. "Echo:*message*").

Run the server using your Python interpreter:

```
python server.py
```

On another window, send messages using the client application:

```
python client.py hello world!
# Echo: hello world!
```

Note: To shut down the server, you can use the key combination: Ctrl+C

Working with Publish/Subscribe Pattern

In the case of publish/subscribe pattern, ZeroMQ is used to establish one or more subscribers, connecting to one or more publishers and receiving continuously what publisher sends (or *seeds*).

A choice to specify a prefix to accept only such messages beginning with it is available with this pattern.

Use-cases:

Publish/subscribe pattern is used for evenly distributing messages across various consumers. Automatic updates for scoreboards and news can be considered as possible areas to use this solution.

Socket type(s) used:

- zmq.PUB
- zmq.SUB

Publisher Example: pub.py

Create a "pub.py" using **nano** (`nano pub.py`) and paste the below contents.

```
import zmq
import time
# ZeroMQ Context
context = zmq.Context()
# Define the socket using the "Context"
sock = context.socket(zmq.PUB)
sock.bind("tcp://127.0.0.1:5680")
id = 0
while True:
    time.sleep(1)
    id, now = id+1, time.ctime()
    # Message [prefix][message]
    message = "1-Update! >> #{id} >> {time}".format(id=id, time=now)
```


PDF Generated by PHPKB Knowledge Base Script

```
    sock.send(message)
    # Message [prefix][message]
    message = "2-Update! >> #{id} >> {time}".format(id=id, time=now)
    sock.send(message)
    id += 1
```

When you are done editing, save and exit by pressing CTRL+X followed with Y.

Subscriber Example: sub.py

Create a "sub.py" using **nano** (`nano sub.py`) and paste the below contents.

```
import zmq
# ZeroMQ Context
context = zmq.Context()
# Define the socket using the "Context"
sock = context.socket(zmq.SUB)
# Define subscription and messages with prefix to accept.
sock.setsockopt(zmq.SUBSCRIBE, "1")
sock.connect("tcp://127.0.0.1:5680")
while True:
    message= sock.recv()
    print message
```

When you are done editing, save and exit by pressing CTRL+X followed with Y.

Note: Using the `.setsockopt(..)`procedure, we are subscribing to receive messages starting with *string* `1` . To receive all, leave it not set (i.e.`""`).

Usage

Our`pub.py`is set to work as a *publisher*, sending two different messages - simultaneously - intended for different subscribers.

Run the publisher to send messages:

```
python pub.py
```

On another window, see the print outs of subscribed content (i.e.`1`):

```
python sub.py!
# 1-Update! >> 1 >> Wed Dec 25 17:23:56 2013
```

Note: To shut down the subscriber and the publisher applications, you can use the key combination: Ctrl+C

Pipelining the Pub./Sub. with Pipeline Patter (Push/Pull)

Very similar in the way it looks to the Publish/Subscribe pattern, the third in line Pipeline pattern comes as a

solution to a different kind of problem: distributing messages upon demand.

Use-cases:

Pipelining pattern can be used in cases where are list of queued items need to be routed (i.e. *pushed* in line) for the one asking for it (i.e. those who *pull*).

Socket type(s) used:

- zmq.PUSH
- zmq.PULL

PUSH Example: manager.py

Create a "manager.py" using **nano** (`nano manager.py`) and paste the below contents.

```
import zmq
import time
# ZeroMQ Context
context = zmq.Context()
# Define the socket using the "Context"
sock = context.socket(zmq.PUSH)
sock.bind("tcp://127.0.0.1:5690")
id = 0
while True:
    time.sleep(1)
    id, now = id+1, time.ctime()
    # Message [id] - [message]
    message = "{id} - {time}".format(id=id, time=now)
    sock.send(message)
    print "Sent: {msg}".format(msg=message)
```

The file `manager.py` will act as a *task allocator*.

PULL Example: worker_1.py

Create a "worker_1.py" using **nano** (`nano worker_1.py`) and paste the below contents.

import zmq

```
# ZeroMQ Context
context = zmq.Context()
# Define the socket using the "Context"
sock = context.socket(zmq.PULL)
sock.connect("tcp://127.0.0.1:5690")
while True:
    message = sock.recv()
    print "Received: {msg}".format(msg=message)
```

The file `worker_1.py` will act as a *task processes* (consumer/worker).

Usage

Our `manager.py` is set to have a role of an allocator of tasks (i.e. a manager), ***PUSH*** ing the items. Likewise, `worker_1.py` set to work as a *worker* instance receives these items, when it's done processing by ***PULL*** ing down the list.

Run the publisher to send messages:

```
python manager.py
```

On another window, see the print outs of subscribed content (i.e.1):

```
python worker_1.py!
# 1-Update! >> 1 >> Wed Dec 25 17:23:56 2013
```

Note: To shut down the subscriber and the publisher applications, you can use the key combination: Ctrl+C

Exclusive Pair Pattern

Exclusive pair pattern implies and allows establishing one-tone sort of communication channels using the `zmq/PAIR` socket type.

Bind Example: bind.py

Create a "bind.py" using **nano** (`nano bind.py`) and paste the below contents.

```
import zmq
# ZeroMQ Context
context = zmq.Context()
# Define the socket using the "Context"
socket = context.socket(zmq.PAIR)
socket.bind("tcp://127.0.0.1:5696")
```

When you are done editing, save and exit by pressing CTRL+X followed with Y.

Connect Example: connect.py

Create a "connect.py" using **nano** (`nano connect.py`) and paste the below contents.

```
import zmq
# ZeroMQ Context
context = zmq.Context()
# Define the socket using the "Context"
socket = context.socket(zmq.PAIR)
```

```
socket.connect("tcp://127.0.0.1:5696")
```

When you are done editing, save and exit by pressing CTRL+X followed with Y.

Usage

You can use the above example to create any bidirectional uni-connection communication applications.

Note: To shut down either, you can use the key combination: Ctrl+C