# Chapter 1

# What is dīvidere?

Dīvidere, latin for "to divide, to seperate" seemed an appropriate package name for a distributed system framework project.

The primary goal of dīvidere is to combine three complementary technologies (Python, ZeroMQ, Protobuf) into a distributed system messaging framework. ZeroMQ will provide a variety of transport mechanisms, Protobuf providing a language-independent, strongly-typed message encoding and Python the means to combine these components into a reusable framework.

# Chapter 2

# ZeroMq

ZeroMq provides the core transport mechanisms used by this framework. We'd highhly recommend referencing the official ZeroMq documentation [1] for more comprehensive material, but for the purposes of this package we will attempt to document sufficient information necessary to use this package.
The communication package provides primitive ZeroMq classes which support byte-stream messaging as the foundation of other more sophisticated packages.

## 2.1 Publish/Subscribe

The publish-subscribe, pub-sub, sometimes referred to as the observer pattern is a software design pattern where producers of messages provide info without knowledge of the recepients. An analogy would be a radio broadcasting station, sending information to an unknown number of recepients. The messaging is one-way, from provider (publisher) to consumer (subscriber). A publisher can choose to produce one specific message, or a series of messages. The subscriber 'subscribes' to a list of messages, afterwhich all produced messages of this 'topic' will be received by the subscriber.

## 2.2 Request/Response

The request-response, or request-reply, provides a sychronous form of message passing. The requester sends a message, then waits for the response. This form of communication enforces a send/receive protocol, failure to comply results in the socket throwing an exception. You may choose to connect multiple response objects to the same requester, if doing so sent messages will be routed one-by-one to each response objects in a round-robin fashion. This pattern allows a worker pool fashion architecture.

[1] Offical ZeroMQ documentation: https://zeromq.org/

# Chapter 3

# Protobuf

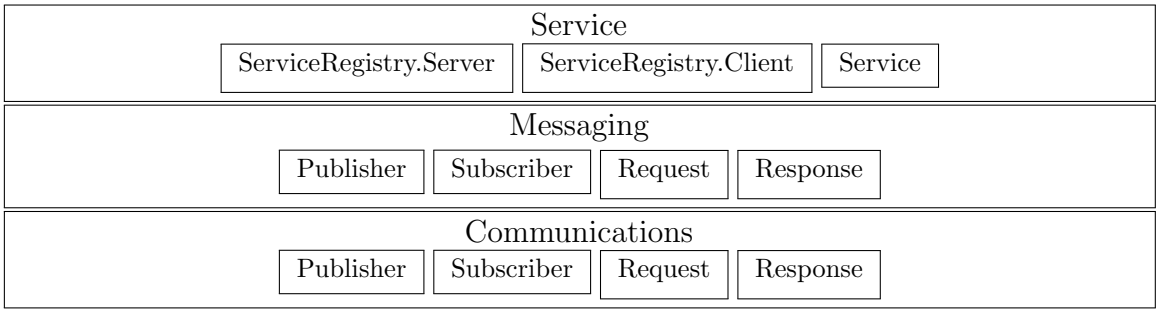
The ZeroMQ transport supports byte-stream and string payloads. Complex messages could be transmitted in JSON form using the communication package but instead we chose to utilize the protobuf encoding/decoding to allow type-safe, language specific messaging contents. Google Protobuf [1] supports a platform-neutral extensible means to define serialing structured data. Messages are defined in a *.proto file, a message compiler converts the proto file into a language-specific (e.g. Python) message library used by the clients.

[1] https://protobuf.dev/

# Chapter 4

# Architecture

Dividere is implemented as a layered architeture, the primary communication layer provided at the *Communications* package, the *Messaging* package providing aggregator classes utilizing the communications classes exchanging Protobuf messages.
These two layers are expected to expand in the future, we also intend on adding higher-level layer(s) with higher-level distributed system abstractions.




Dividere implements a layered architecture approach, more primitive abstractions located at the lower layers, with specialized abstractions atop. Upper layers utilizing lower layer componenents.

## 4.1 Communications

The communications layer focuses on providing string-based messaging components with generalized debugging visibility. Each consumer component allows blocking and time-out blocking message retrieval interfaces. Most of these components provide a light-weight facade to ZeroMQ components.

## 4.2 Messaging

This layer mirrors many of the components from the communications layer with a subtle difference, components in this layer utilize protobuf messaging protocol rather than string-based messages. This layer is intended to provide multi-language integration support.

## 4.3 Service

The service layer provides service-based abstactions, including a 'Service' abstract class that registers with the centralized, server-based, name-service.

# Chapter 5

# Examples

simplePubSub.py

```
#!/usr/bin/python3
import dividere
import clientMsgs_pb2 as clientMsgs
import time

Port=5555
pub=dividere.messaging.Publisher('tcp://*:%d'%(Port))
sub=dividere.messaging.Subscriber('tcp://localhost:%d'%(Port))
time.sleep(2); #--delay to address 'late joiner'

msg=clientMsgs.msg01()
msg.field1='abcd'
pub.send(msg)
got=sub.recv()
assert(got==msg)

#--destroy pub/sub objects to free resources and terminate threads
pub=None
sub=None
```

# Chapter 6

# Reference

```
|      specified endpoint (e.g. 'tcp://*:5555')
|      Refer to ZMQ documentation for details on available transport
|      and syntax of endpoint.
|
|  send(self, msg)
|      Publish the specified message (expected sequence of bytes)
|
|  ----------------------------------------------------------------------
|  Methods inherited from Connector:
|
|  __del__(self)
|      Performs cleanup for all allocated resources;
|      disable monitoring, wait for monitoring thread completes,
|      close the socket and close the context
|
|  ----------------------------------------------------------------------
|  Static methods inherited from Connector:
|
|  registerSocketMonitoring(sock)
|      Creates a monitoring thread for the specified socket,
|      starts the thread and returns the thread id to the caller
|      which allows joining on the thread post stopping monitoring
|      Note: Used internally to class(es), not intended for external usage
|
|  socketEventMonitor(monitorSock)
|      Background threading callback, supports monitoring the
|      specified socket via a background thread and logs state
|      changes of the socket for debugging purposes.
|      Monitors the socket until monitoring is terminated
|      via object destructor (e.g. obj = None)
|      Note: Used internally to class(es), not intended for external usage
```

```
 |
 |  ----------------------------------------------------------------------
 |  Data descriptors inherited from Connector:
 |
 |  __dict__
 |      dictionary for instance variables (if defined)
 |
 |  __weakref__
 |      list of weak references to the object (if defined)

class Request(Connector)
 |  Request(endPointList)
 |
 |  First part of a Request/Response connection pair.  Request object
 |  initiates all messages, response object sends message response.
 |  Failure to adhere to this sender protocol will result in exception
 |  being thrown.
 |  Note: this pairing allows for 1-N cardinality, one request connection
 |        object sending to N-response objects.  When configured like this
 |        the recipient of any message is routed in a round-robin fashion
 |        to one response object
 |
 |  Method resolution order:
 |      Request
 |      Connector
 |      builtins.object
 |
 |  Methods defined here:
 |
 |  __init__(self, endPointList)
 |      Allocate all resources to support the object;
 |      create a socket, register it for monitoring, and connect
 |      it to the specified endpoint
 |
 |  recv(self)
 |      Wait for and return the incoming message.
 |
 |  send(self, msg)
 |      Send the specified message out the socket channel.
 |      Message consists of a stream of bytes.
 |
 |  wait(self, timeOutMs)
 |      Wait for a message to arrive within the specified timeout, return
 |      true/false representing whether a message is available
 |
 |  ----------------------------------------------------------------------
```

```
 |  Methods inherited from Connector:
 |
 |  __del__(self)
 |      Performs cleanup for all allocated resources;
 |      disable monitoring, wait for monitoring thread completes,
 |      close the socket and close the context
 |
 |  ----------------------------------------------------------------------
 |  Static methods inherited from Connector:
 |
 |  registerSocketMonitoring(sock)
 |      Creates a monitoring thread for the specified socket,
 |      starts the thread and returns the thread id to the caller
 |      which allows joining on the thread post stopping monitoring
 |      Note: Used internally to class(es), not intended for external usage
 |
 |  socketEventMonitor(monitorSock)
 |      Background threading callback, supports monitoring the
 |      specified socket via a background thread and logs state
 |      changes of the socket for debugging purposes.
 |      Monitors the socket until monitoring is terminated
 |      via object destructor (e.g. obj = None)
 |      Note: Used internally to class(es), not intended for external usage
 |
 |  ----------------------------------------------------------------------
 |  Data descriptors inherited from Connector:
 |
 |  __dict__
 |      dictionary for instance variables (if defined)
 |
 |  __weakref__
 |      list of weak references to the object (if defined)

class Response(Connector)
 |  Response(endPoint)
 |
 |  Second part of a Request/Response connection pair.  Request object
 |  initiates all messages, response object sends message response.
 |  Failure to adhere to this sender protocol will result in exception
 |  being thrown.
 |
 |  Method resolution order:
 |      Response
 |      Connector
 |      builtins.object
 |
```

```
|  Methods defined here:
|
|  __init__(self, endPoint)
|      Allocate all resources to support the object;
|      create a socket, register it for monitoring, and connect
|      it to the specified endpoint
|
|  recv(self)
|      Wait for and return the incoming message.
|
|  send(self, msg)
|      Send the specified message out the socket channel
|      Message consists of a stream of bytes.
|
|  wait(self, timeOutMs)
|      Wait for a message to arrive within the specified timeout, return
|      true/false representing whether a message is available
|
|  ----------------------------------------------------------------------
|  Methods inherited from Connector:
|
|  __del__(self)
|      Performs cleanup for all allocated resources;
|      disable monitoring, wait for monitoring thread completes,
|      close the socket and close the context
|
|  ----------------------------------------------------------------------
|  Static methods inherited from Connector:
|
|  registerSocketMonitoring(sock)
|      Creates a monitoring thread for the specified socket,
|      starts the thread and returns the thread id to the caller
|      which allows joining on the thread post stopping monitoring
|      Note: Used internally to class(es), not intended for external usage
|
|  socketEventMonitor(monitorSock)
|      Background threading callback, supports monitoring the
|      specified socket via a background thread and logs state
|      changes of the socket for debugging purposes.
|      Monitors the socket until monitoring is terminated
|      via object destructor (e.g. obj = None)
|      Note: Used internally to class(es), not intended for external usage
|
|  ----------------------------------------------------------------------
|  Data descriptors inherited from Connector:
|
```

```
 |  __dict__
 |      dictionary for instance variables (if defined)
 |
 |  __weakref__
 |      list of weak references to the object (if defined)

class Subscriber(Connector)
 |  Subscriber(endPoint, topic='')
 |
 |  This class creates a subscriber socket at the specified endpoint.
 |  This is the sub in the Pub/Sub pattern.  By default, a subscriber
 |  object will listen for all messages, but can be filtered by specifying
 |  a topic(s); either by specifying a topic during the initializer or
 |  calling subscribe() after object creation
 |
 |  Method resolution order:
 |      Subscriber
 |      Connector
 |      builtins.object
 |
 |  Methods defined here:
 |
 |  __init__(self, endPoint, topic='')
 |      Allocate base class resources, create SUB socket, start
 |      socket debug monitoring and connect the socket to the
 |      specified endpoint (e.g. 'tcp://localhost:5555')
 |      Subscribes to the specified topic, by default the object
 |      will receive all messages.
 |      Refer to ZMQ documentation for details on available transport
 |      and syntax of endpoint.
 |
 |  recv(self)
 |      Wait for next message to arrive and return it to the
 |      caller.
 |
 |  subscribe(self, topic)
 |      Allows subscribing to additional topics (beyond the one
 |      specified in the constructor)
 |
 |  wait(self, timeOutMs)
 |      Wait for a message to arrive within the specified timeout, return
 |      true/false representing whether a message is available
 |
 |  ----------------------------------------------------------------------
 |  Methods inherited from Connector:
 |
```

```
 |  __del__(self)
 |      Performs cleanup for all allocated resources;
 |      disable monitoring, wait for monitoring thread completes,
 |      close the socket and close the context
 |
 |  ----------------------------------------------------------------------
 |  Static methods inherited from Connector:
 |
 |  registerSocketMonitoring(sock)
 |      Creates a monitoring thread for the specified socket,
 |      starts the thread and returns the thread id to the caller
 |      which allows joining on the thread post stopping monitoring
 |      Note: Used internally to class(es), not intended for external usage
 |
 |  socketEventMonitor(monitorSock)
 |      Background threading callback, supports monitoring the
 |      specified socket via a background thread and logs state
 |      changes of the socket for debugging purposes.
 |      Monitors the socket until monitoring is terminated
 |      via object destructor (e.g. obj = None)
 |      Note: Used internally to class(es), not intended for external usage
 |
 |  ----------------------------------------------------------------------
 |  Data descriptors inherited from Connector:
 |
 |  __dict__
 |      dictionary for instance variables (if defined)
 |
 |  __weakref__
 |      list of weak references to the object (if defined)

FILE
```