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

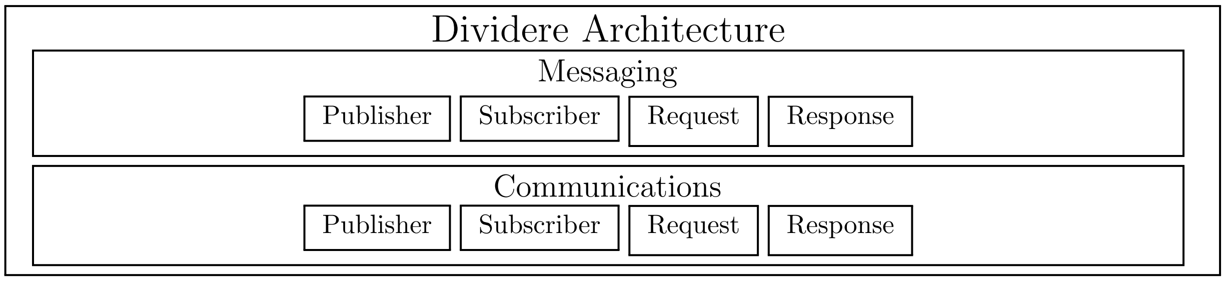
The ZeroMQ transport supports byte-stream and string payloads. Complex messages could be transmitted in JSON form using the communication package but instead we chose to utilize the protobuf encoding/decoding to allow type-safe, language specific messaging contents. Google Protobuf [1] supports a platform-neutral 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.



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
```