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

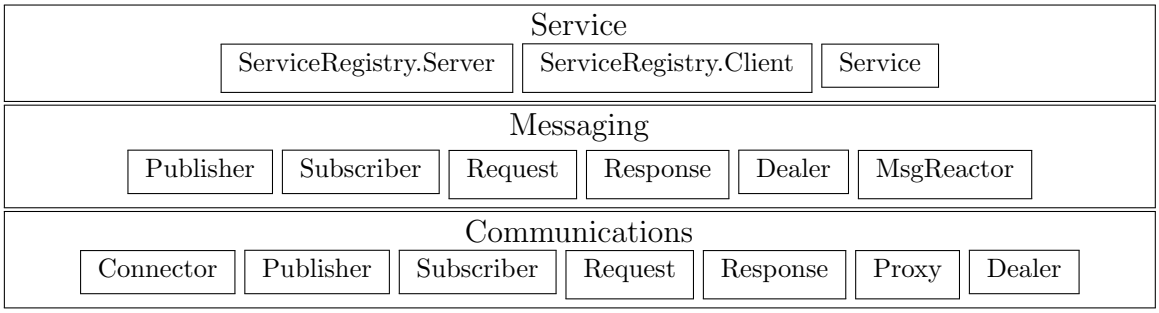
The ZeroMQ transport supports byte-stream and string payloads. Complex messages could be transmitted in JSON form using the communication package but instead we chose to utilize the protobuf encoding/decoding to allow type-safe, language specific messaging contents. Google Protobuf [1] supports a platform-neutral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

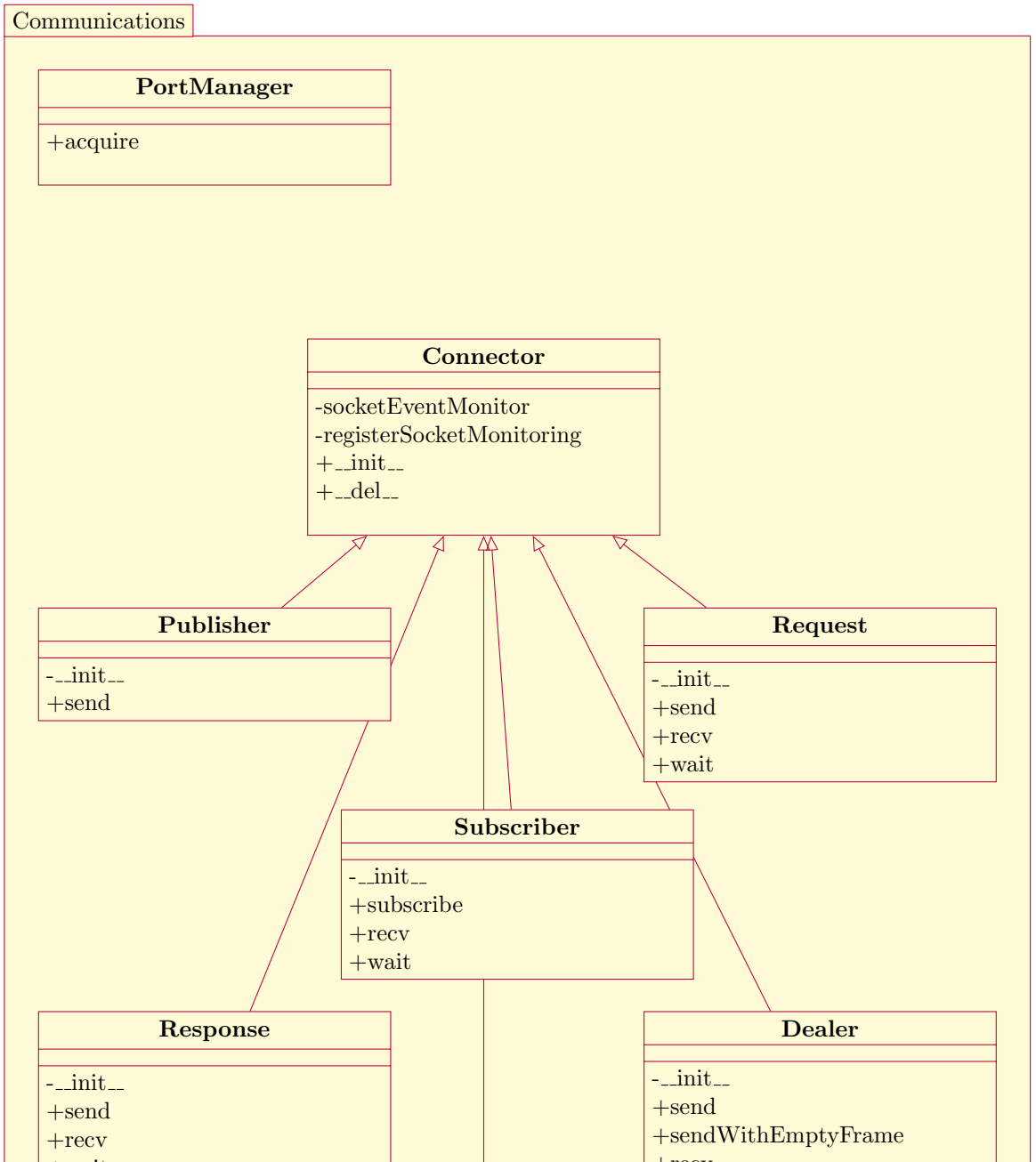
This layer mirrors many of the components from the communications layer with a subtle difference, components in this layer utilize protobuf messaging protocol rather than string-based messages. This layer is intended to provide multi-language integration support.

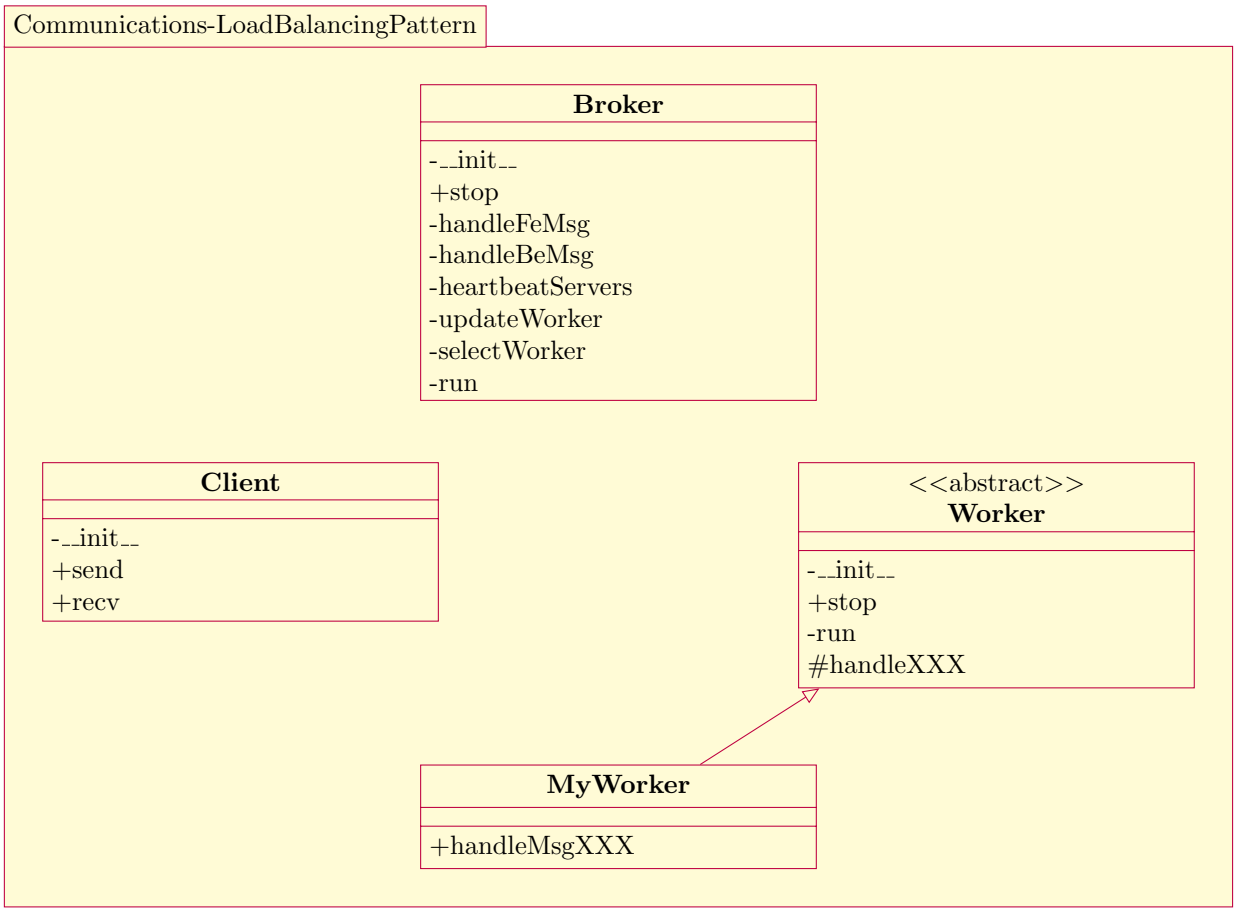
4.3 Service

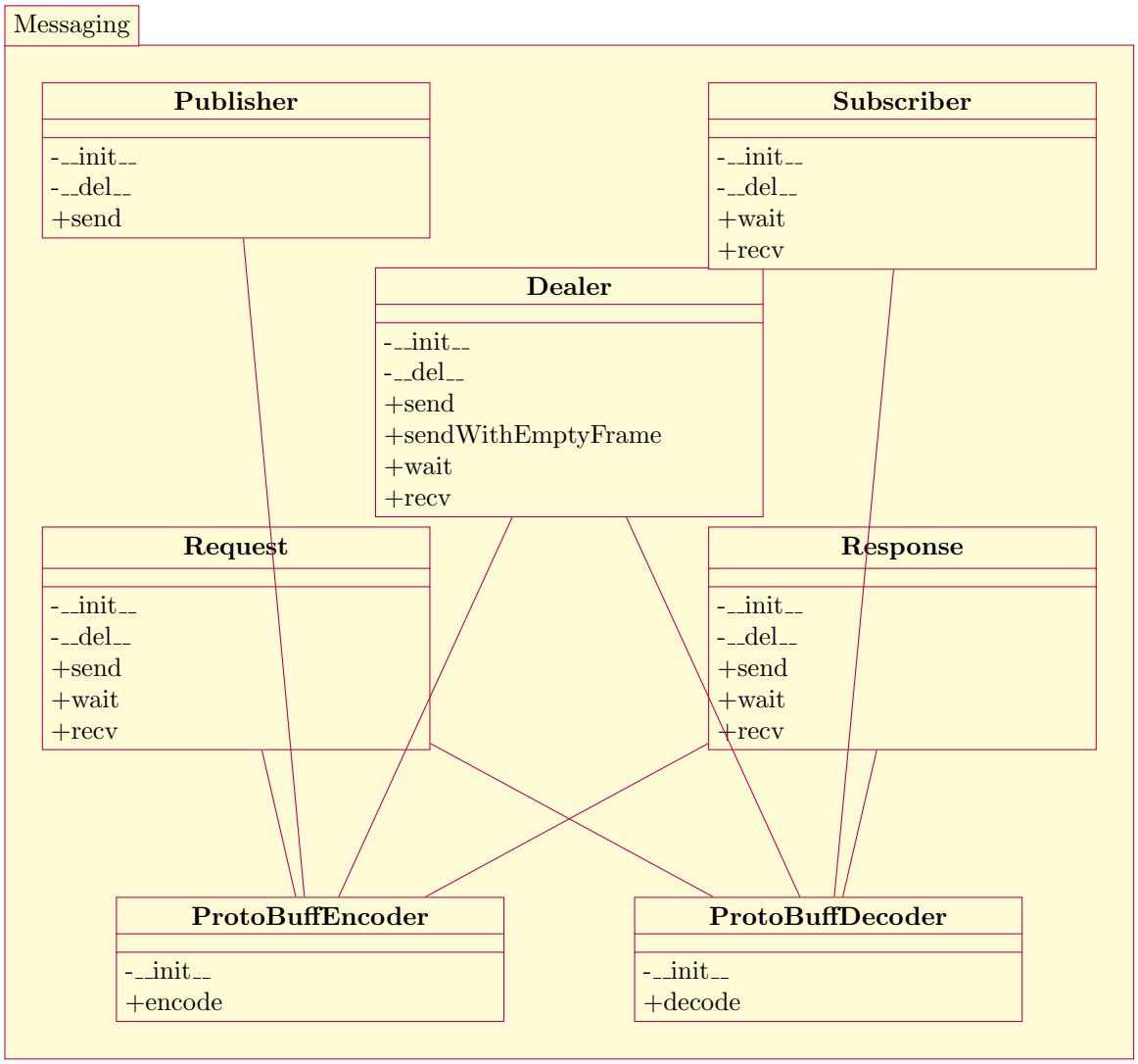
The service layer provides service-based abstactions, including a 'Service' abstract class that registers with the centralized, server-based, name-service.

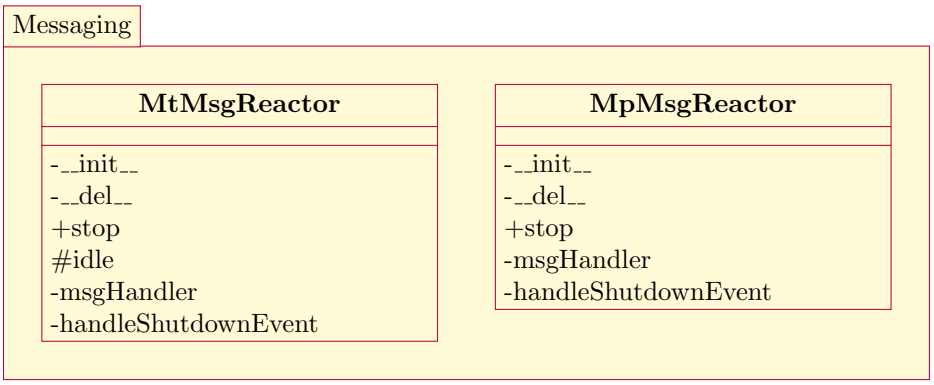
Chapter 5

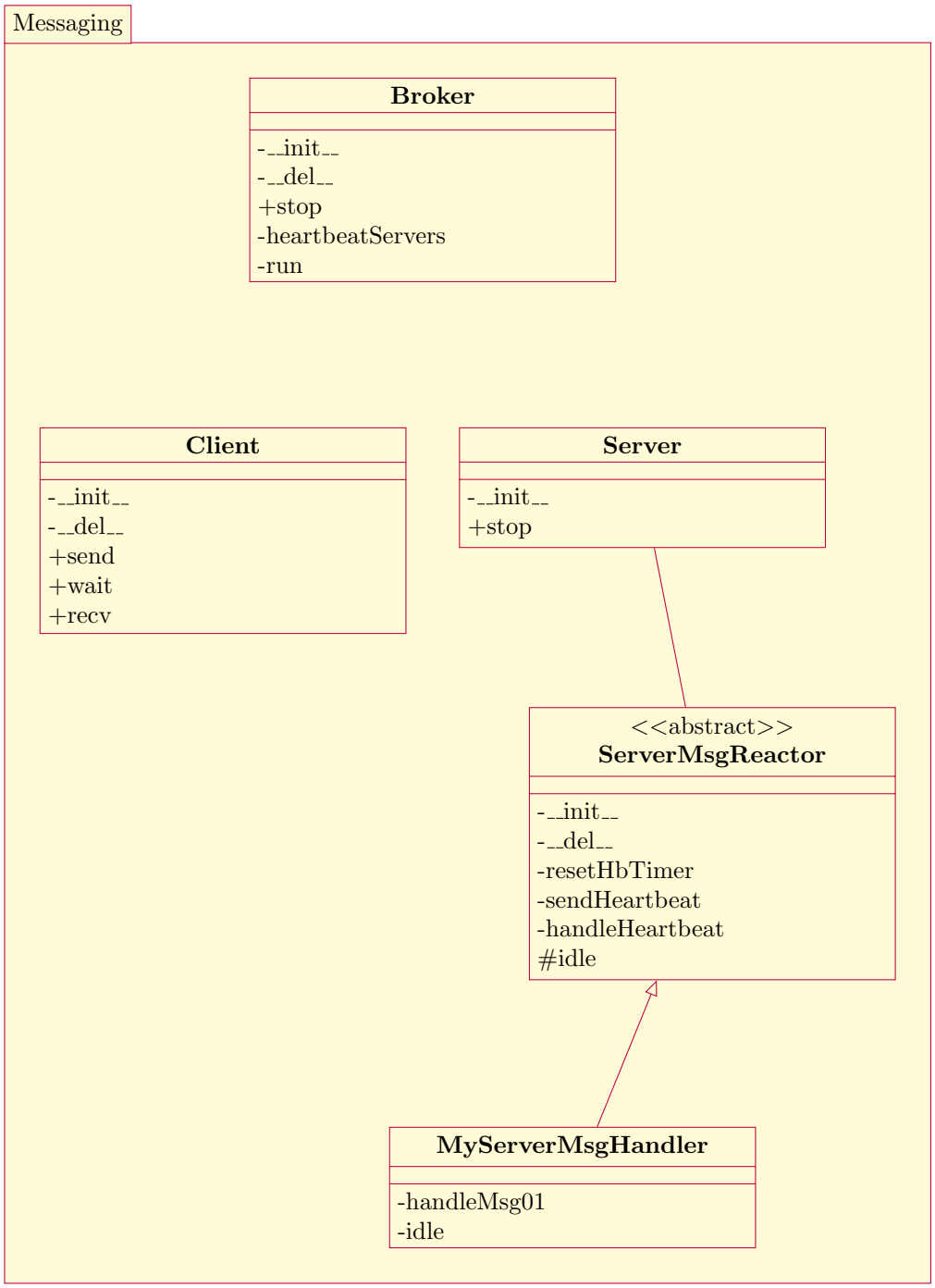
Class Design

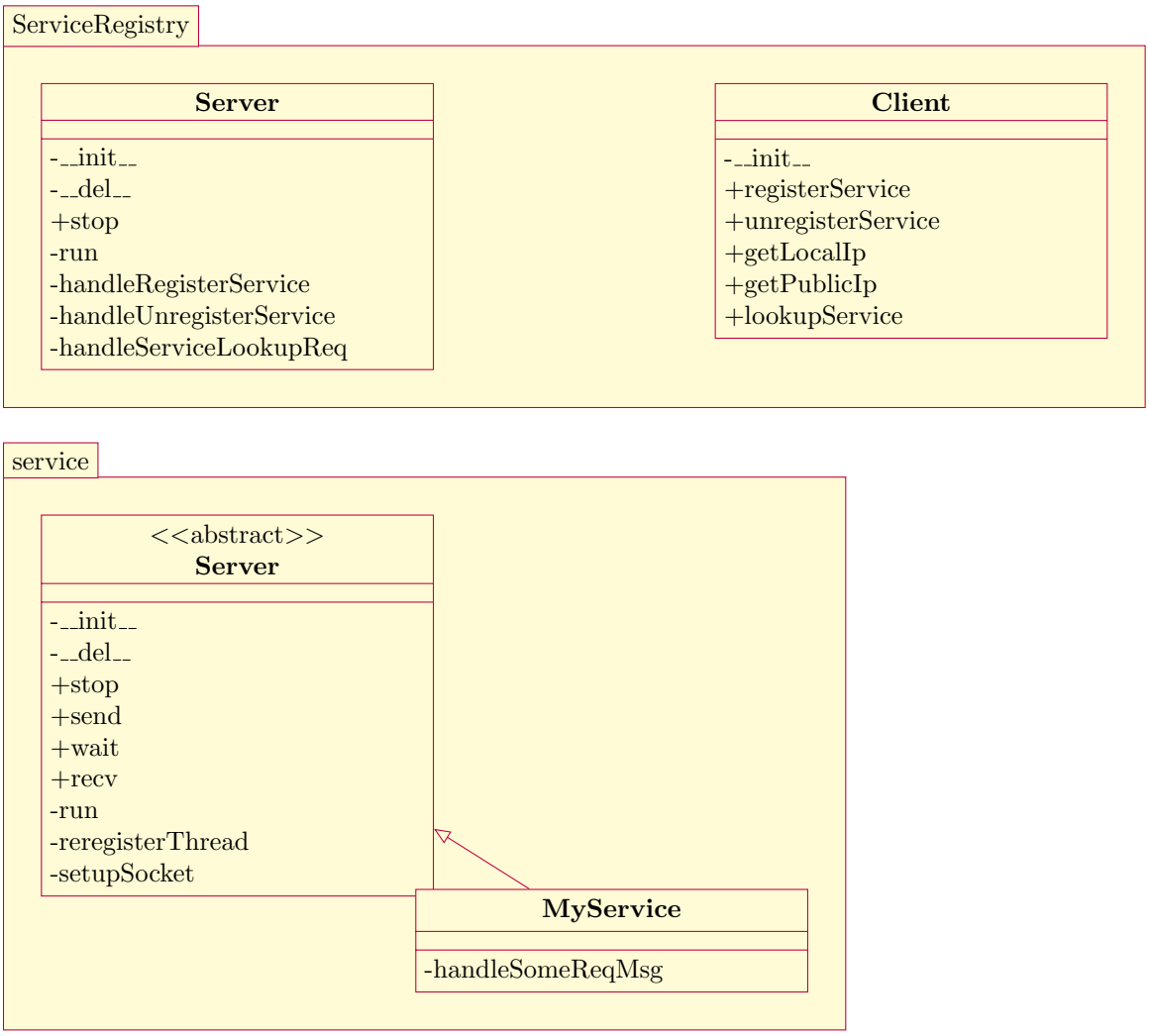



Communications-LoadBalancingPattern
Broker
-__init__
+stop
-handleFeMsg
-handleBeMsg
-heartbeatServers
-updateWorker
-selectWorker
-run
Client
-__init__
+send
+recv
<<abstract>>
Worker
-__init__
+stop
-run
#handleXXX
MyWorker
+handleMsgXXX


Messaging
Publisher
-__init__
-__del__
+send
Subscriber
-__init__
-__del__
+wait
+recv
Dealer
-__init__
-__del__
+send
+sendWithEmptyFrame
+wait
+recv
Request
-__init__
-__del__
+send
+wait
+recv
Response
-__init__
-__del__
+send
+wait
+recv
ProtoBuffEncoder
-__init__
+encode
ProtoBuffDecoder
-__init__
+decode


Messaging
MtMsgReactor
-__init__
-__del__
+stop
#idle
-msgHandler
-handleShutdownEvent
MpMsgReactor
-__init__
-__del__
+stop
-msgHandler
-handleShutdownEvent


Messaging
Broker
-__init__
-__del__
+stop
-heartbeatServers
-run
Client
-__init__
-__del__
+send
+wait
+recv
Server
-__init__
+stop
<<abstract>>
ServerMsgReactor
-__init__
-__del__
-resetHbTimer
-sendHeartbeat
-handleHeartbeat
#idle
MyServerMsgHandler
-handleMsg01
-idle


ServiceRegistry
Server
-__init__
-__del__
+stop
-run
-handleRegisterService
-handleUnregisterService
-handleServiceLookupReq
Client
-__init__
+registerService
+unregisterService
+getLocalIp
+getPublicIp
+lookupService
service
<<abstract>>
Server
-__init__
-__del__
+stop
+send
+wait
+recv
-run
-reregisterThread
-setupSocket
MyService
-handleSomeReqMsg

Chapter 6

Examples

simplePubSub.py

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

Port=dividere.connection.PortManager.acquire()
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)
reply=sub.recv()
print("reply: %s"%(reply))
assert(reply==msg)

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

msgReactor.py

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

class MyMsgReactor(dividere.messaging.MtMsgReactor):
  def handleMsg01(self, obj, msg):
    print("got msg01 msg: %s"%(str(msg)))
    msg.field1=msg.field1[::-1]
    obj.send(msg)

  def handleMsg02(self, obj, msg):
    print("got msg02 msg: %s"%(str(msg)))

  def initThread(self):
    pass

class MyMsgReactor(dividere.messaging.MpMsgReactor):
  def handleMsg01(self, obj, msg):
    print("got msg01 msg: %s"%(str(msg)))
    msg.field1=msg.field1[::-1]
    obj.send(msg)
    print("sent response %s"%(msg))

  def handleMsg02(self, obj, msg):
    print("got msg02 msg: %s"%(str(msg)))

  def initThread(self):
    pass


def test00():
  fePort=dividere.connection.PortManager.acquire()
  bePort=dividere.connection.PortManager.acquire()
  mh=MyMsgReactor([dividere.messaging.Response('tcp://*:%d'%(fePort)),
                   dividere.messaging.Subscriber('tcp://localhost:%d'%(be

  req=dividere.messaging.Request('tcp://localhost:%d'%(fePort))
  msg=clientMsgs.Msg01()
  msg.field1='hello'
  req.send(msg)

  pub=dividere.messaging.Publisher('tcp://*:%d'%(bePort))

  msg2=clientMsgs.Msg02()
  msg2.field1='some published event'
  time.sleep(1); #--accomodate late joiner
  pub.send(msg2)

  reply=req.recv()
  assert(reply.field1==msg.field1[::-1])
  print(reply)
  mh.stop()
```

Chapter 7

Reference

```
Help on module connection:

NAME
    connection

CLASSES
    builtins.object
        Connector
            Dealer
            Proxy
            Publisher
            Request
            Response
            Subscriber
        LoadBalancingPattern
        PortManager

    class Connector(builtins.object)
     |  This abstract class defines the interfaces and structures
     |  for ZMQ socket-based derived classes.  This class provides
     |  the ZMQ context and socket event monitoring useful for
     |  debugging socket state changes.
     |  The socket monitoring is conducted by an independent thread,
     |  which is terminated/joined at object termination.
     |
     |  Methods defined here:
     |
     |  __del__(self)
     |      Performs cleanup for all allocated resources;
     |      disable monitoring, wait for monitoring thread completes,
     |      close the socket and close the context
```

```
 |
 |  __init__(self)
 |      Creates resources used in base classes and defines expected
 |      structure to be used in derived classes.
 |
 |  ----------------------------------------------------------------------
 |  Static methods defined here:
 |
 |  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 defined here:
 |
 |  __dict__
 |      dictionary for instance variables (if defined)
 |
 |  __weakref__
 |      list of weak references to the object (if defined)

class Dealer(Connector)
 |  Dealer(endPointList)
 |
 |  Dealer 'generally' is a replacement for Request/Response objects without
 |  the strict send/receive protocol.  Use of dealer objects allow asynchronous
 |  messaging, like sending N messages rather than the strict send/recv protocol.
 |
 |  Method resolution order:
 |      Dealer
 |      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)
|      Inbound message could be routed, or unrouted, if routed return
|      the identifier vector and message content.  The final frame
|      will be the message content, the preceeding frames will be
|      routing identifiers (maybe multiples if message routed thru multiple
|      router sockets).
|      Returned value will _either_ be message payload, or tuple with routing id
|      vector + message payload
|
|  send(self, msg)
|      Dealer socket must be capable of sending routed or unrouted messages,
|      for example; client-side messages to anonymous workers may be unrouted,
|      and worker responses may be routed.  All depending on the communications
|
|  sendWithEmptyFrame(self, msg)
|      Send message but with preceeding empty identity frame, used to emulate
|      request message protocol (e.g. Dealer-Response connections)
|
|  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 LoadBalancingPattern(builtins.object)
 |  Load balancing pattern; broker, worker, client
 |
 |  Data descriptors defined here:
 |
 |  __dict__
 |      dictionary for instance variables (if defined)
 |
 |  __weakref__
 |      list of weak references to the object (if defined)
 |
 |  ----------------------------------------------------------------------
 |  Data and other attributes defined here:
 |
 |  Broker = <class 'connection.LoadBalancingPattern.Broker'>
 |      Load balancing broker, workers register to backend port, clients through
 |      frontend.  Broker routes in round-robin fashion.  Broker and workers
 |      exchange heartbeats to recover when worker, or broker, fails and restarts.
 |
 |
 |  Client = <class 'connection.LoadBalancingPattern.Client'>
 |      Front-end component for pattern, reliable request-reply mechanism
 |       by utilizing retry policy
 |
 |
 |  Worker = <class 'connection.LoadBalancingPattern.Worker'>
 |      Worker, active object, connects to broker and maintains heartbeating protoc
 |      Specialization is intended by deriving from this abstract class.

class PortManager(builtins.object)
 |  Singleton supports acquiring an available port for service
 |  and communications objects.
```

```
 |
 |  Static methods defined here:
 |
 |  acquire()
 |      Find next available port, as per the os, by creating a
 |      temporary socket which is assigned an available port
 |      number, then close the socket and return the port for
 |      use.
 |      Note, port acquisition should support multi-threaded
 |      clients.
 |
 |  ----------------------------------------------------------------------
 |  Data descriptors defined here:
 |
 |  __dict__
 |      dictionary for instance variables (if defined)
 |
 |  __weakref__
 |      list of weak references to the object (if defined)

class Proxy(Connector)
 |  Proxy(fePort, bePort)
 |
 |  Proxy abstraction defines a router/dealer pairing to allow
 |  async req/rep client connections.
 |
 |  Method resolution order:
 |      Proxy
 |      Connector
 |      builtins.object
 |
 |  Methods defined here:
 |
 |  __init__(self, fePort, bePort)
 |      Front-end utilizes the base class socket_ attribute, adds a backend
 |      socket.  Binds to two known ports
 |
 |  run(self)
 |      Loop until signaled to stop, wait for an event
 |      for a specified time-out, to prevent blocking calls,
 |      then route inbound messages from one to the other socket
 |
 |  stop(self)
 |      Signal the active thread it should terminate, wait for
 |      the thread to halt and close out derived class resources
 |
```

```
 |  ----------------------------------------------------------------------
 |  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 Publisher(Connector)
 |  Publisher(endPoint)
 |
 |  This class creates a publisher socket at the specified endpoint.
 |  This is the pub in the Pub/Sub pattern.
 |
 |  Method resolution order:
 |      Publisher
 |      Connector
 |      builtins.object
 |
 |  Methods defined here:
```

```
 |
 |  __init__(self, endPoint)
 |      Allocate base class resources, create PUB socket, start
 |      socket debug monitoring and connect the socket to the
 |      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)

FUNCTIONS
    Lock = allocate_lock(...)
        allocate_lock() -> lock object
        (allocate() is an obsolete synonym)

        Create a new lock object. See help(type(threading.Lock())) for
        information about locks.

DATA
    logger = <Logger connection (ERROR)>

FILE
Help on module messaging:

NAME
    messaging

CLASSES
    builtins.object
        Dealer
        LoadBalancingPattern
        MpMsgReactor
        MtMsgReactor
        ProtoBuffDecoder
        ProtoBuffEncoder
        Publisher
```

```
    Request
    Response
    Subscriber

class Dealer(builtins.object)
 |  Dealer(endPoint)
 |
 |  General replacement for Request/Response components, but relaxes
 |  the strict send/receive protocol.  This component support more
 |  asynchronous messaging by allowing multiple send/recv functionality.
 |
 |  Methods defined here:
 |
 |  __del__(self)
 |      Free all allocated object resources
 |
 |  __init__(self, endPoint)
 |      Allocate all necessary resources, including socket and encoder/decoder
 |      pair.  All transported communications will be done in the form of a
 |      message envelope
 |
 |  recv(self)
 |      Return value _may_ be a single message, or a tuple (id,msg)
 |      depending on usage.  Routed messages (e.g. one thru a router,
 |      may include the 'identity' (route) of the message so it can be
 |      routed back to the originating sender.
 |
 |  send(self, msg)
 |      Encode message into envelope container, convert it to
 |      a byte stream and send out wire via the connector
 |
 |  sendWithEmptyFrame(self, msg)
 |      Send message but with preceeding empty identity frame, used to emulate
 |      request message protocol (e.g. Dealer-Response connections)
 |
 |  wait(self, timeOutMs)
 |      Wait for a message to arrive within the specified timeout, return
 |      true/false representing whether a message is available
 |
 |  ----------------------------------------------------------------------
 |  Data descriptors defined here:
 |
 |  __dict__
 |      dictionary for instance variables (if defined)
 |
 |  __weakref__
```

```
 |      list of weak references to the object (if defined)

class LoadBalancingPattern(builtins.object)
 |  General pattern; client(s), broker, server(s).  Clients send requests to broke
 |  determines available server, forwards requests and routes response back to ori
 |  server.
 |
 |  Data descriptors defined here:
 |
 |  __dict__
 |      dictionary for instance variables (if defined)
 |
 |  __weakref__
 |      list of weak references to the object (if defined)
 |
 |  ----------------------------------------------------------------------
 |  Data and other attributes defined here:
 |
 |  Broker = <class 'messaging.LoadBalancingPattern.Broker'>
 |      Broker acts as the intermediatry between clients and servers,
 |      requests coming in from clients are routed to available servers.
 |      The broker is also responsible for managing 'active' servers, meaning
 |      keeping track of servers that are responsive and available for inbound
 |      requests.
 |
 |
 |  Client = <class 'messaging.LoadBalancingPattern.Client'>
 |      Lazy-Pirate Reliable Request/Response; tracks last message sent
 |      and if response isn't received within a time-out, message is resent
 |      using a max retry policy.
 |
 |
 |  Server = <class 'messaging.LoadBalancingPattern.Server'>
 |      Server abstraction, connects and HB's with broker.

class MpMsgReactor(builtins.object)
 |  MpMsgReactor(obj)
 |
 |  Multi-Process Msg Reactor
 |  Multi-process (MP) abstraction, rather than multi-threaded, to take full advan
 |  processor cores.  Note, the constructor provides a string list of messaging co
 |  rather than an actual list of objects because they must be created in the back
 |  different than threaded usage.
 |  Derived classes are intended to specialize initialization method that is invok
 |  background process to initialize resources (e.g. def initThread(self))
 |
```

```
 |  Methods defined here:
 |
 |  __del__(self)
 |      Free resources created by client process
 |
 |  __init__(self, obj)
 |      Initialize necessary components, shutdown pub/sub uses tcp endpoints to support multi
 |      communications.
 |
 |  handleShutdownEvent(self, obj, msg)
 |      Set the done flag, this is done from the thread to avoid need for necessary guards
 |
 |  msgHandler(self, objList, shutdownEndPt)
 |      Background process callback, iterates over specified message object list
 |      looking for available messages, then invokes the associated callback
 |      based on message name specialized by the derived class.  Inbound shutdown
 |      event is handled by this class, which flags completion of the task.
 |
 |  stop(self)
 |      Signal termination and await process completion.
 |
 |  ----------------------------------------------------------------------
 |  Data descriptors defined here:
 |
 |  __dict__
 |      dictionary for instance variables (if defined)
 |
 |  __weakref__
 |      list of weak references to the object (if defined)

class MtMsgReactor(builtins.object)
 |  MtMsgReactor(obj)
 |
 |  Multi-Threaded Msg Reactor
 |  Abstraction to support active-thread which listens to a vector of a
 |  varying consumer messaging objects (e.g. Sub, Response, ...), decoding
 |  the incoming message and calling a specialized hander method (provided mostly
 |  by derived classes).
 |
 |  Methods defined here:
 |
 |  __del__(self)
 |      Deallocate all messaging objects, which in-turn terminates the zmq contexts
 |
 |  __init__(self, obj)
 |      Spawn an independent thread which monitors the specified consumer message
```

```
 |      objects, also append an additional object to support multi-threaded signall
 |      to support halting the thread when no longer needed.
 |      (ipc pub/sub is used to signal thread termination)
 |
 |  handleShutdownEvent(self, obj, msg)
 |      Set the done flag, this is done from the thread to avoid need for necessary
 |
 |  idle(self)
 |      Method called between processing messages, meant to be extended by child cl
 |      when necessary
 |
 |  msgHandler(self)
 |      This method encapsulates the 'active object' logic, while 'not done'
 |      poll/wait for an inbound message from any messaging object in the list
 |      if a message exists, grab it and call a specialized message handler functio
 |      (based on message name), provide the messaging object it arrived on
 |      to allow handler to choose to send reply (for compliant messaging objects l
 |
 |  stop(self)
 |      Signal thread to complete, wait for it to complete
 |
 |  ----------------------------------------------------------------------
 |  Data descriptors defined here:
 |
 |  __dict__
 |      dictionary for instance variables (if defined)
 |
 |  __weakref__
 |      list of weak references to the object (if defined)

class ProtoBuffDecoder(builtins.object)
 |  This class suports taking in a user protobuf message and encode/pack
 |  into a container message for transport.  This is one end of a encode/decode
 |  sequence used when sending a user message through a socket while allowing
 |  a variety of messages to be sent thru a shared socket channel.
 |  This is one end of the encode/decode sequence; encoding done at the sending
 |  end, decoding at the receiving end.
 |
 |  Methods defined here:
 |
 |  __init__(self)
 |      Initialize self.  See help(type(self)) for accurate signature.
 |
 |  decode(self, msgEnv)
 |      Extract the user message from the specified container message
 |      and return it to the caller.
```

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

class ProtoBuffEncoder(builtins.object)
 |  This class suports taking in a user protobuf message and encode/pack
 |  into a container message for transport.  This is one end of a encode/decode
 |  sequence used when sending a user message through a socket while allowing
 |  a variety of messages to be sent thru a shared socket channel.
 |  This is one end of the encode/decode sequence; encoding done at the sending
 |  end, decoding at the receiving end.
 |
 |  Methods defined here:
 |
 |  __init__(self)
 |      Initialize object resources
 |
 |  encode(self, msg)
 |      Encapsulate the specified message into a container message for
 |      transport and return it to the caller
 |
 |  ----------------------------------------------------------------------
 |  Data descriptors defined here:
 |
 |  __dict__
 |      dictionary for instance variables (if defined)
 |
 |  __weakref__
 |      list of weak references to the object (if defined)

class Publisher(builtins.object)
 |  Publisher(endPoint)
 |
 |  Similar functionality to the Publish/Subscriber pairing in the connection
 |  module, differing in the expected user message being sent.  The messaging
 |  module specializes in sending/receiving protobuf-based messages.
 |
 |  Methods defined here:
 |
 |  __del__(self)
```

```
 |      Free allocated object resources
 |
 |  __init__(self, endPoint)
 |      Create a publisher connection and encoder
 |
 |  send(self, msg)
 |      Encode message into envelope container, convert it to
 |      a byte stream and send out wire via the connector
 |
 |  ----------------------------------------------------------------------
 |  Data descriptors defined here:
 |
 |  __dict__
 |      dictionary for instance variables (if defined)
 |
 |  __weakref__
 |      list of weak references to the object (if defined)

class Request(builtins.object)
 |  Request(endPoint)
 |
 |  Similar functionality to the Request/Response pairing in the connection
 |  module, differing in the expected user message being sent.  The messaging
 |  module specializes in sending/receiving protobuf-based messages.
 |
 |  Methods defined here:
 |
 |  __del__(self)
 |      Free allocated object resources
 |
 |  __init__(self, endPoint)
 |      Create a request connection and encoder
 |
 |  recv(self)
 |      Retrieve byte stream from response, parse byte stream into envelope
 |       message, then decode and return the contained user message
 |
 |  send(self, msg)
 |      Encode message into envelope container, convert it to
 |      a byte stream and send out wire via the connector
 |
 |  wait(self, timeOutMs)
 |      Wait for a message to arrive within the specified timeout, return
 |      true/false representing whether a message is available
 |
 |  ----------------------------------------------------------------------
```

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

class Response(builtins.object)
 |  Response(endPoint)
 |
 |  Similar functionality to the Request/Response pairing in the connection
 |  module, differing in the expected user message being sent.  The messaging
 |  module specializes in sending/receiving protobuf-based messages.
 |
 |  Methods defined here:
 |
 |  __del__(self)
 |      Free all allocated object resources
 |
 |  __init__(self, endPoint)
 |      Allocate all necessary resources, socket and encoder/decoder pair.
 |
 |  recv(self)
 |      Retrieve byte stream from requester, parse byte stream into envelope
 |       message, then decode and return the contained user message
 |
 |  send(self, msg)
 |      Encode message into envelope container, convert it to
 |      a byte stream and send out wire via the connector
 |
 |  wait(self, timeOutMs)
 |      Wait for a message to arrive within the specified timeout, return
 |      true/false representing whether a message is available
 |
 |  ----------------------------------------------------------------------
 |  Data descriptors defined here:
 |
 |  __dict__
 |      dictionary for instance variables (if defined)
 |
 |  __weakref__
 |      list of weak references to the object (if defined)

class Subscriber(builtins.object)
 |  Subscriber(endPoint, msgSubList=[])
```

```
 |
 |  Similar functionality to the Publish/Subscriber pairing in the connection
 |  module, differing in the expected user message being sent.  The messaging
 |  module specializes in sending/receiving protobuf-based messages.
 |
 |  Methods defined here:
 |
 |  __del__(self)
 |      Free all allocated object resources
 |
 |  __init__(self, endPoint, msgSubList=[])
 |      Allocate all necessary resources, subscribe to messages.
 |      If message subscription list is empty, subscribe to all messages
 |      otherwise subscribe to the specified messages exclusively
 |      create subscriber object and decoder components
 |
 |  recv(self)
 |      Retrieve byte stream from subscriber, parse byte stream into envelope
 |       message, then decode and return the contained user message
 |
 |  wait(self, timeOutMs)
 |      Wait for a message to arrive within the specified timeout, return
 |      true/false representing whether a message is available
 |
 |  ----------------------------------------------------------------------
 |  Static methods defined here:
 |
 |  topicId(msg)
 |      Translate a protobuf message into a topic name
 |      (the beginning of the string coming across the 'wire')
 |      used to subscribe to specific message(s)
 |      Note: expected usage is internal to the module, not
 |      intended for external use
 |
 |  ----------------------------------------------------------------------
 |  Data descriptors defined here:
 |
 |  __dict__
 |      dictionary for instance variables (if defined)
 |
 |  __weakref__
 |      list of weak references to the object (if defined)

DATA
    logger = <Logger messaging (ERROR)>
```

```
FILE
Help on module registry:

NAME
    registry

CLASSES
    builtins.object
        ServiceRegistry

    class ServiceRegistry(builtins.object)
     |  Primarily namespace, server-side class used for instantiating a
     |  nameservice, client-side for performing registration and service
     |  lookup.
     |
     |  Data descriptors defined here:
     |
     |  __dict__
     |      dictionary for instance variables (if defined)
     |
     |  __weakref__
     |      list of weak references to the object (if defined)
     |
     |  ----------------------------------------------------------------------
     |  Data and other attributes defined here:
     |
     |  Client = <class 'registry.ServiceRegistry.Client'>
     |      Instantiate new object, open port to name service
     |
     |
     |  Server = <class 'registry.ServiceRegistry.Server'>
     |      Server-side implementation; establish a well-defined port for
     |      incoming registration and lookup requests.  Open the incoming port
     |      and wait for incoming messages in an independent thread.

FILE
Help on module service:

NAME
    service

CLASSES
    builtins.object
        Service

    class Service(builtins.object)
```

```
|  Abstract base class for services, registers service name with name
|  registry and establishes a req/rep socket for incoming messaging.
|  Derived classes are intended to provide 'def handleXXX(self, msg)'
|  methods for expected incoming requests.
|
|  Methods defined here:
|
|  __del__(self)
|      Force stopping threads if the object is terminated
|
|  __init__(self)
|      Find an available port within port range [5100,6000], create
|      incoming socket with the port, register the service (e.g. derived class nar
|      and port with the name service, then begin waiting for an processing inbour
|      messages in an active thread.
|
|  recv(self)
|      Get the next message from the socket, blocks indefnitely, use wait()
|      to avoid blocking.
|
|  reregisterThread(self)
|      This thread supports reregistration in the case that
|      a name service abruptly terminates, is restarted, and
|      notifies services to re-register
|
|  run(self)
|      Loop waiting for message, call associated message handler (which is respons
|      for sending response message).  Periodically check for signal to terminate
|      the thread.
|
|  send(self, msg)
|      Send message through socket
|
|  setupSocket(self)
|      Loop thru the port range looking for an available port, once
|      finding one register the service and port.  Throw exception
|      if you fail to find an available port
|
|  stop(self)
|      Signal thread to halt.
|
|  wait(self, timeOutMs)
|      Wait for an inbound message within the specified timeout, return bool
|      indicating message was received
|
|  ----------------------------------------------------------------------
```

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

FILE
```