REEM

Release v1.0

Trishul Nagenalli, Dr. Kris Hauser

May 23, 2019

CONTENTS

REEM (Redis Extendable Efficient Middleware) is a centralized middleware package for robotic communication. It is designed to be a single-package solution for passing information anywhere in the robot while emphasizing ease of use and efficiency.

To make it easy, we chose to model information as a nested data structure that closely resembles python dictionaries. To the user, working with a database feels like working with a python dictionary. Out of the box, REEM supports communicating all native python types and numpy arrays.

To make it fast, we used Redis (an in-memory key-value database) running ReJSON (enabling Redis to store JSON data) as a central information store. To get maximum performance, we give users the power to control exactly how information is passed between the local program and Redis by defining their own encoder/decoder objects.

REEM currently offers two communication paradigms: - get/set database - publish-subscribe

To install the python package (and its dependencies), run

```
pip install reem
```

CHAPTER

ONE

SET UP TUTORIAL

In REEM, data is passed between client programs and a centralized Redis server. This tutorial will demonstrate how to set up the server and connect to it with a REEM client. Both the server and client will run on the local machine.

Requirements:

- Python 3
- Linux/macOS (ReJSON requirement, though you can run ReJSON with Docker on Windows)

1.1 Server

This section goes through how to set up a server. REEM runs on Redis and requires the ReJSON module. We will install both and check that they are working.

1.1.1 Redis

The following script will download and build Redis with supporting packages from source inside a folder called `database-server`. REEM has been tested with Redis version 5.0.4. You may want to pull the latest version of Redis in the future. Change the versioning in the script appropriately

DO NOT install Redis through `apt-get install redis-server` This will install Redis 3 which does not support modules. You will not be able to run REEM.

Once you download and build Redis from source, you will need to access two executables: `redis-server` and `redis-cli`. The former is the executable that launches a redis-server. The latter is a useful command line interface (cli) that allows for easy testing. The executables are located at

`database-server/redis-5.0.4/src/redis-server`

`database-server/redis-5.0.4/src/redis-cli`

The script below gives them aliases to make things easier. Note that these aliases will disappear when the terminal closes.

```
mkdir database-server
cd database-server
wget http://download.redis.io/releases/redis-5.0.4.tar.gz
tar xzf redis-5.0.4.tar.gz
cd redis-5.0.4/deps
make hiredis lua jemalloc linenoise
cd ..
make
alias redis-server=$PWD/src/redis-server
```

(continues on next page)

(continued from previous page)

```
alias redis-cli=$PWD/src/redis-cli
cd ..
```

Check that the version of Redis you have is 5.0.x by running `redis-server --version` Now, check that the redis server will boot. Run `redis-server` in your terminal. The redis server will take over your terminal.

Open up another terminal and run `redis-cli`. The CLI will take over that terminal and your prompt should look like `127.0.0.1:6379>` Execute a basic set and get with Redis, ensuring the output looks similar to the output below:

```
127.0.0.1:6379> SET key 1
OK
127.0.0.1:6379> GET key
"1"
127.0.0.1:6379>
```

Congratulations! You have successfully installed and ran Redis. Shutdown the Redis server (issue the `shutdown` command in the cli) and exit the cli.

1.1.2 ReJSON

ReJSON is a third party module developed for Redis developed by Redis Labs. It introduces a JSON datatype to Redis that is not available in standard Redis. REEM relies on it for serializable data.

Starting from inside the `database-server` folder, continuing from the Redis installation script, the following will build ReJSON from source.

```
git clone https://github.com/RedisLabsModules/redisjson.git
cd redisjson
make
cd ..
```

The above script produces an compiled library file at `database-server/redisjson/src/rejson.so`. Redis needs to be told to use that library. You can tell Redis that by starting a server with a configuration file. Download this example configuration file and place it inside `database-server`.

Some details about this configuration file:

- Line 46 (in the modules section) says `loadmodule redisjson/src/rejson.so` specifying the compiled library for rejson
- Line 71 (in the network section) says `bind 127.0.0.1` to bind only to the local host network interface.

If you later want to make this redis server accessible on a network, you must change line 71 to bind to that interface too. For example if the computer hosting the redis server has an ip address `10.0.0.1` on the network, this line should become `bind 127.0.0.1 10.0.0.1` so that it binds to the local interface and the network interface.

Let's test the ReJSON installation. Run `redis-server redis.conf`. This will start the Redis server with ReJSON. Open another terminal and run `redis-cli`. Be sure you can execute the following in that redis-cli prompt

```
127.0.0.1:6379> JSON.SET foo . 0
OK
```

1.2 Client

Before you begin this part of the turtorial, make sure a redis server is available for a client to connect to. If a server is not already running, run `redis-server redis.conf` in a terminal and leave that terminal be.

Client machines connect to the server purely through Python with the REEM client. Install REEM and it's dependencies with the below command

```
pip3 install reem
```

Copy the below into a file and run it:

```
from reem.connection import RedisInterface
from reem.datatypes import KeyValueStore
import numpy as np
import time

interface = RedisInterface(host="localhost")
interface.initialize()
server = KeyValueStore(interface)

# Set a key and read it and its subkeys
server["foo"] = {"number": 100.0, "string": "REEM"}
print("Reading Root  : {}".format(server["foo"].read()))
print("Reading Subkey: {}".format(server["foo"]["number"].read()))

# Set a new key that didn't exist before to a numpy array
server["foo"]["numpy"] = np.random.rand(3,4)
time.sleep(0.0001)  # Needed on ubuntu machine for numpy set to register?
print("Reading Root  : {}".format(server["foo"].read()))
print("Reading Subkey: {}".format(server["foo"]["numpy"].read()))
```

The output should appear something like the below

```
Reading Root  : {'number': 100, 'string': 'REEM'}
Reading Subkey: 100
Reading Root  : {'number': 100, 'string': 'REEM', 'numpy': array([[0.41949741, 0.
↪40785201, 0.70637666, 0.1809309 ],
       [0.37884759, 0.70176005, 0.14115555, 0.82246663],
       [0.24243882, 0.86587402, 0.19852017, 0.21833667]])}
Reading Subkey: [[0.41949741 0.40785201 0.70637666 0.1809309 ]
 [0.37884759 0.70176005 0.14115555 0.82246663]
 [0.24243882 0.86587402 0.19852017 0.21833667]]
```

The code connects to a Redis server and `set` s a dictionary with basic number and string data. It then reads and prints that data. Next, it sends a numpy array to Redis and reads that back as well. It uses a KeyValueStore object to do all this. Learn more about it in the next section.

Congratulations! You have got REEM working on your machine! Continue to the next section to see what it can do.

CHAPTER

TWO

BASIC USAGE

This page explains how to use database and publish/subscribe paradigms with REEM.

2.1 Initialization

Before any information can be passed to a Redis server, we need to specify how to contact the server. A `RedisInterface` object is meant to represent a connection to a specific server. Instantiate it and call initialize before attaching any datatypes to it. You must specify the host as the IP address of the server running Redis (or localhost). If no host is provided, **the default argument for host is localhost**

```
from reem.connection import RedisInterface
interface = RedisInterface(host="localhost")
interface.initialize()
```

2.2 Key Value Store

The `KeyValueStore` object is meant to be your way of interacting with Redis as a nested database server. You should treat a `KeyValueStore` object as though it were a python dictionary that can contain native python types and numpy arrays. When you set something inside this "dictionary", the corresponding entry will be set in Redis. Reading the "dictionary" will read the corresponding entry in Redis.

The `KeyValueStore` is instantiated with a `RedisInterface` object, identifying what Redis server it is connected to.

```
from reem.datatypes import KeyValueStore
server = KeyValueStore(interface)
```

The below code illustrates:

- To set an item in Redis, the syntax is identical to that setting a path in a Python dictionary
- To get an item from Redis, the syntax is the same as a dictionary's but you must call `.read()` on the final path.

```
data = {'number': 1000, 'string': 'REEM'}
server["foo"] = flat_data

bar = server["foo"].read()
# Sets bar = {'number': 1000, 'string': 'REEM'}

bar = server["foo"]["number"].read()
# Sets bar = 1000
```

Limitations

1. Cannot use non-string Keys

```
server["foo"] = {0:"zero", 1:"one"} # Not Okay
server["foo"] = {"0":"zero", "1":"one"} # Okay
```

REEM assumes all keys are strings to avoid having to parse JSON keys to determine if they are strings or numbers.

2. Cannot have a list with non-serializable types.

```
server["foo"] = {"bar":[np.arange(3), np.arange(4)]} # Not Okay
server["foo"] = {"bar":[3, 4]} # Okay
```

REEM does not presently check lists for non serializable types. We hope to allow this in a future release. For now, we ask you substitute the list with a dictionary

```
server["foo"] = {"bar":[np.arange(3), np.arange(4)]} # Not Okay
server["foo"] = {"bar":{"arr1": np.arange(3), "arr2": np.arange(4)}} # Okay
```

2.3 Publish/Subscribe

Publishing and subscribing is implemented with a single type of publisher and two types of subscribers.

2.3.1 Publisher

Publishers are implemented with the `PublisherSpace` class and are instantiated with a `RedisInterface`. You may treat a `PublishSpace` like an python dictionary that you CANNOT read.

```
from reem.datatypes import PublishSpace
publisher = PublishSpace(interface)
```

When you set something inside this "dictionary" the publisher broadcasts a message indicating what path was updated. All subscribers listening to that path are notified and act accordingly.

```
data = {"image": np.random.rand(640, 480, 3), "id": 0}

# publishes raw_image
publisher["raw_image"] = data

# publishes raw_image.id
publisher["raw_image"]["id"] = 1
```

All limitations that apply to `KeyValueStore` apply to `PublishSpace` as well. `PublishSpace` is a subclass of `KeyValueStore`.

2.3.2 Subscribers

Subscribes listen to a key on the Redis Server and will act based on changes to that key OR its sub-keys. For example a subscriber to the key "raw_image" will be notified if "raw_image" is freshly uploaded by a publisher and if the path "raw_image.id" is updated.

A subscriber's `.listen()` method must be called for it to start listening to Redis updates.

Subscribing has two implementations

Silent Subscribers

A silent subscriber acts like a local variable that mimics the data in Redis underneath the key indicated by its channel. It will silently update as fast as it can without notifying the user that an update occurred. Use it if you would like a variable that just keeps the latest copy of Redis information at all times.

The `SilentSubscriber` is initialized with a channel name and an interface. The channel represents the path inside the RedisServer this subscriber should listen to. Initialization is as below

```
from reem.datatypes import SilentSubscriber
subscriber = SilentSubscriber(channel="silent_channel", interface=interface)
subscriber.listen()
```

The below code illustrates how to read data from a subscriber.

```
publisher["silent_channel"] = {"number": 5, "string":"REEM"}
time.sleep(0.01)

foo = subscriber["number"].read()
# foo = 5
foo = subscriber.value()
# foo = {"number": 5, "string":"REEM"}

publisher["silent_channel"] = 5
time.sleep(0.01)

foo = subscriber.value()
# foo = 5
```

Note: The `.read()` method does not go to Redis but copies the value at that path in the local variable. This is faster than the `.read()` method used by the `KeyValueStore` which does go to Redis.

Callback Subscribers

Callback Subscribers listen to a key in Redis and execute a user-specified function when an update occurs. They are instantiated with an interface, a channel name, a function, and a dictionary specifying keyword arguments to the function.

Instantiation is as below

```
def callback(data, updated_path, foo):
 print("Foo = {}".format(foo))
 print("Data = {}".format(data))

# Initialize a callback subscriber
subscriber = CallbackSubscriber(channel="callback_channel",
                                interface=interface,
                                callback_function=callback,
                                kwargs={"foo":5})
subscriber.listen()
```

The Callback Function

The callback function must have `data` and `updated_path` as it's first two arguments. When a publisher sets a key, `data` gives the entire updated data structure below the key and `updated_path` tells what path was updated. Further arguments can be passed as keyword arguments set during the instantiation of subscriber.

If the publisher executes

```
publisher["callback_channel"] = {"number": 5, "string": "REEM"}
publisher["callback_channel"]["number"] = 6
```

The subscriber program will have the following output:

```
Foo = 5
Updated Path = callback_channel
Data = {'number': 6, 'string': 'REEM'}
Foo = 5
Updated Path = callback_channel.number
Data = {'number': 6, 'string': 'REEM'}
```

CHAPTER

THREE

ADVANCED USAGE

This section explains less common functionality of REEM.

3.1 Custom Datatypes

REEM is designed to be customizable. Out of the box, it supports transferring native python types and numpy arrays. You can, however, define how any type of data is stored in Redis using a `Ship` object.

Inside the module, `reem.ships` is the abstract class `SpecialDatatypeShip`. If you define your own ship, you must subclass `SpecialDatatypeShip` and fill in the methods. The class's documentation is below

class reem.ships.**SpecialDatatypeShip**

check_fit(*value*)
Determine if this ship will handle `value`

This method returns true if `value` is data that this ship is supposed to handle. If this ship handled all numpy arrays, it would check if `value`'s type is a numpy array.

Parameters **value** – object to check

Returns: True if ship will handle `value`

write(*key*, *value*, *client*)
Write `value` to Redis at the specified `key` using `client`

Given a Redis client, execute any number of needed commands to store the `value` in Redis. You are required to use the key given for REEM to find it. If you must store multiple pieces of information, use a Redis Hash which acts like a one level dictionary.

Parameters

- **key** (*str*) – The Redis key name this ship must store data under
- **value** – The value to write into Redis
- **client** – A ReJSON Redis Client pipeline

Returns: None

read(*key*, *client*)
Retrieve necessary information from Redis

Given a Redis client, execute ONE command to retrieve all the information you need to rebuild the data that was stored in `write` from Redis. This method should execute the command that allows you to retrieve all data stored under key

Parameters

- **key** (*str*) – a keyname that contains data stored by `write`
- **client** – A ReJSON Redis Client pipeline

Returns: None

interpret_read(*responses*)

Translate Redis data into a local object

Redis will reply to you with something according to what read command you executed in `read`. This method takes whatever Redis replied with and turns it into an object identical to what was initially passed to `write` as value.

Parameters **responses** – Redis's reply data based on `read` method

Returns: An object identical to what was initially written to Redis.

get_label()

Return a unique string identifier

This method should return a string that uniquely identifies this ship. REEM will use it to determine what ship to use to decode data that is already stored in Redis.

Returns the string identifier

Return type str

To use a ship, include it as an argument when creating a `RedisInterface` object.

```
interface = RedisInterface(host="localhost", ships=[CustomShip()])
```

Numpy Arrays

Numpy Arrays are stored in Redis through ships. If you want to keep the default ship for numpy arrays when including your custom ships, you must include the default numpy ship in your list of ships.

```
interface = RedisInterface(host="localhost", ships=[reem.ships.NumpyShip(),␣
↪CustomShip()])
```

See the implementation of the Numpy Ship below

```
class NumpyShip(SpecialDatatypeShip):
    def check_fit(self, value):
        return type(value) in [np.array, np.ndarray]

    def write(self, key, value, client):
        client.hset(key, "arr", bytes(memoryview(value.data)))
        client.hset(key, "dtype", str(value.dtype))
        client.hset(key, "shape", str(value.shape))
        client.hset(key, "strides", str(value.strides))

    def get_label(self):
        return "default_numpy_handler"

    def read(self, key,  client):
        client.hgetall(key)

    def interpret_read(self, responses):
        hash = responses[0]
        dtype = eval("np.{}".format(hash[b'dtype'].decode('utf-8')))
```

(continues on next page)

(continued from previous page)

```
        shape = hash[b'shape'].decode("utf-8")[1:-1]
        shape = tuple([int(s) for s in shape.split(",") if len(s) > 0])
        arr = np.frombuffer(hash[b'arr'], dtype)
        arr = np.reshape(arr, shape)
        return arr
```

CHAPTER

FOUR

EXAMPLES

4.1 Image Processing System

Below is an example of what an image processing system might look like. In this robot, there are three components. The robot's computation flow is as below

1. A camera takes an image and posts it to Redis
2. A computer processes that image and posts the result to Redis. Here, the process is just to compute the mean value of the image.
3. An actuator reads the result of the computation and does something with it. Here, it just logs it.

All the code and each component's logs can be found in the repository

4.1.1 Camera

```
from reem.datatypes import PublishSpace
from reem.connection import RedisInterface
import numpy as np
import time
import logging

# Logging Configuration
logging.basicConfig(
    format="%(asctime)20s %(filename)30s:%(lineno)3s  %(funcName)20s()
↪%(levelname)10s     %(message)s",
    filename="camera.log",
    filemode='w')
logger = logging.getLogger("script")
logger.setLevel(logging.INFO)

TIME_TO_RUN = 5.0  # seconds
start_time = time.time()


# --------------------------- Main -----------------------------------


interface = RedisInterface(host="localhost")
pspace = PublishSpace(interface=interface)

image_count = 0
```

(continues on next page)

(continued from previous page)

```
while time.time() < start_time + TIME_TO_RUN:
    image = np.random.rand(640, 480, 3)
    data = {
        "image": image,
        "images_sent": image_count,
        "time_stamp": time.time(),
    }
    pspace["raw_image"] = data
    logger.info("Published Image {}".format(image_count))
    image_count += 1
```

4.1.2 Processor

```
from reem.datatypes import PublishSpace, CallbackSubscriber
from reem.connection import RedisInterface
import numpy as np
import time
import logging

# Logging Configuration
logging.basicConfig(
    format="%(asctime)20s %(filename)30s:%(lineno)3s  %(funcName)20s()
↪%(levelname)10s     %(message)s",
    filename="processor.log",
    filemode='w')
logger = logging.getLogger("script")
logger.setLevel(logging.INFO)

TIME_TO_RUN = 5.0  # seconds


# ---------------------------- Main -----------------------------------


interface = RedisInterface(host="localhost")
pspace = PublishSpace(interface=interface)


def callback(data, updated_path):
    pspace["processed_info"] = {
        "mean": np.mean(data["image"]),
        "time_stamp": time.time(),
        "images_sent": data["images_sent"]
    }
    logger.info("Processed image {}".format(data["images_sent"]))


subscriber = CallbackSubscriber(
    channel="raw_image",
    interface=interface,
    callback_function=callback,
    kwargs={}
)

subscriber.listen()
```

(continues on next page)

(continued from previous page)

```
time.sleep(TIME_TO_RUN)
```

4.1.3 Actuator

```
from reem.datatypes import PublishSpace, CallbackSubscriber
from reem.connection import RedisInterface
import time
import logging

# Logging Configuration
logging.basicConfig(
    format="%(asctime)20s %(filename)30s:%(lineno)3s  %(funcName)20s()
↪%(levelname)10s     %(message)s",
    filename="actuator.log",
    filemode='w')
logger = logging.getLogger("script")
logger.setLevel(logging.INFO)

TIME_TO_RUN = 5.0  # seconds


# ---------------------------- Main -----------------------------------


interface = RedisInterface(host="localhost")
pspace = PublishSpace(interface=interface)


def callback(data, updated_path):
    logger.info("Processed image {}".format(data["images_sent"]))


subscriber = CallbackSubscriber(
    channel="processed_info",
    interface=interface,
    callback_function=callback,
    kwargs={}
)

subscriber.listen()
time.sleep(TIME_TO_RUN)
```

4.2 Arm Actuator

This examples tries to mimic a system where one computer is responsible for actuating motors at a specific frequency while the set point is controlled by another computer at a different frequency.

We have implemented in two ways - using a database paradigm and a publish/subscribe paradigm.

All the code and each component's logs can be found in the repository

4.2.1 Database

Controller

```
from reem.datatypes import KeyValueStore, CallbackSubscriber
from reem.connection import RedisInterface
import time
import logging

# Logging Configuration
logging.basicConfig(
    format="%(asctime)20s %(filename)30s:%(lineno)3s  %(funcName)20s()
↪%(levelname)10s     %(message)s",
    filename="controller_kvs.log",
    filemode='w')
logger = logging.getLogger("script")
logger.setLevel(logging.INFO)

TIME_TO_RUN = 5.0  # seconds
start_time = time.time()


# --------------------------- Main -----------------------------------


interface = RedisInterface(host="localhost")
kvs = KeyValueStore(interface=interface)

set_frequency = 100  # Hz
set_period = 1.0/set_frequency

while time.time() < start_time + TIME_TO_RUN:
    next_iteration = time.time() + set_period
    command = time.time()
    kvs["set_point"] = command
    logger.info("Wrote Set Point: {}".format(command))
    time.sleep(max(0.0, next_iteration - time.time()))
```

Actuator

```
from reem.datatypes import KeyValueStore
from reem.connection import RedisInterface
import time
import logging

# Logging Configuration
logging.basicConfig(
    format="%(asctime)20s %(filename)30s:%(lineno)3s  %(funcName)20s()
↪%(levelname)10s     %(message)s",
    filename="actuator_kvs.log",
    filemode='w')
logger = logging.getLogger("script")
logger.setLevel(logging.INFO)

TIME_TO_RUN = 5.0  # seconds
start_time = time.time()

# --------------------------- Main -----------------------------------
```

(continues on next page)

(continued from previous page)

```
interface = RedisInterface(host="localhost")
kvs = KeyValueStore(interface)

polling_frequency = 1000  # Hz
polling_period = 1.0/polling_frequency

while time.time() < start_time + TIME_TO_RUN:
    next_iteration = time.time() + polling_period
    command = kvs["set_point"].read()
    logger.info("Read Set Point: {}".format(command))
    time.sleep(max(0.0, next_iteration - time.time()))
```

4.2.2 Publish/Subscribe

Controller

```
from reem.datatypes import PublishSpace
from reem.connection import RedisInterface
import time
import logging

# Logging Configuration
logging.basicConfig(
    format="%(asctime)20s %(filename)30s:%(lineno)3s  %(funcName)20s()
↪%(levelname)10s     %(message)s",
    filename="controller_silent_subcsriber.log",
    filemode='w')
logger = logging.getLogger("script")
logger.setLevel(logging.INFO)

TIME_TO_RUN = 5.0  # seconds
start_time = time.time()


# ---------------------------- Main -----------------------------------


interface = RedisInterface(host="localhost")
pspace = PublishSpace(interface=interface)

set_frequency = 100  # Hz
set_period = 1.0/set_frequency

while time.time() < start_time + TIME_TO_RUN:
    next_iteration = time.time() + set_period
    command = time.time()
    pspace["command"] = command
    logger.info("Published Set Point: {}".format(command))
    time.sleep(max(0.0, next_iteration - time.time()))
```

Actuator

```
from reem.datatypes import SilentSubscriber
from reem.connection import RedisInterface
import time
import logging

# Logging Configuration
logging.basicConfig(
    format="%(asctime)20s %(filename)30s:%(lineno)3s  %(funcName)20s()
↪%(levelname)10s     %(message)s",
    filename="actuator_silent_subscriber.log",
    filemode='w')
logger = logging.getLogger("script")
logger.setLevel(logging.INFO)

TIME_TO_RUN = 5.0  # seconds
start_time = time.time()

# --------------------------- Main -----------------------------------

interface = RedisInterface(host="localhost")
subscriber = SilentSubscriber(channel="command", interface=interface)
subscriber.listen()

frequency = 1000  # Hz
period = 1.0/frequency

while time.time() < start_time + TIME_TO_RUN:
    next_iteration = time.time() + period
    command = subscriber.value()
    logger.info("Read Set Point: {}".format(command))
    time.sleep(max(0.0, next_iteration - time.time()))
```

CHAPTER

FIVE

PERFORMANCE

The below documents REEM Performance. See the repository for the full source code and more information.

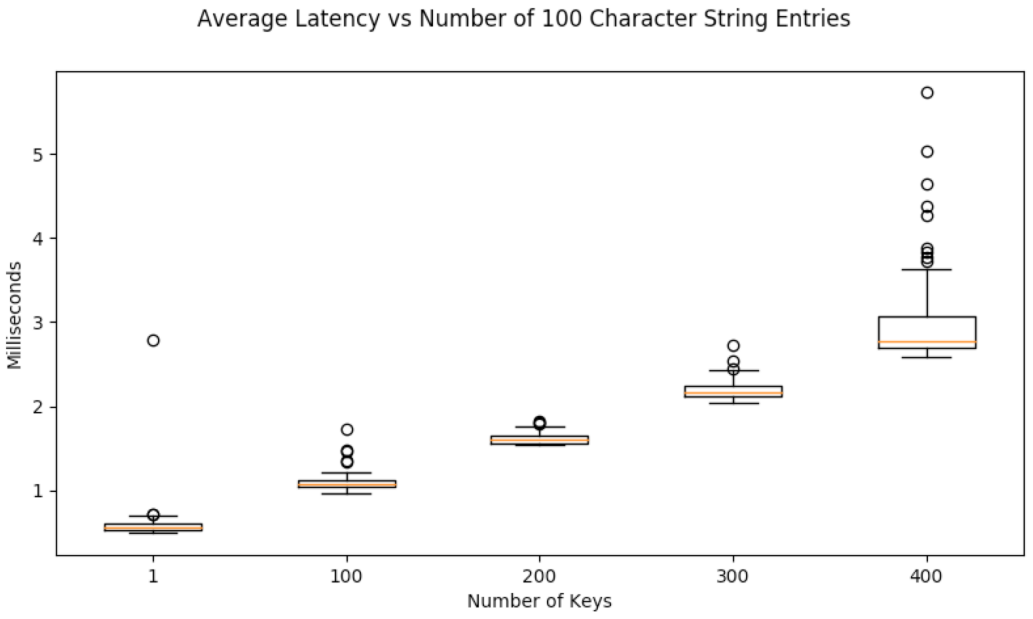
5.1 Data Transfer Rates

Both the database and pubsub paradigms use the same methods for data transfer. The below tests apply to both even though it was conducted in the database paradigm.

5.1.1 Number of Entries vs Latency

String Data

In the below test, a dictionary with X string entries was written to Redis. One hundred trials were conducted. The box-plots of their latency is below. Latency grows linearly with the amount of data sent. The source code to generate this plot is below.




```
def key_growth_strings():
    info = {"title": "Average Latency vs Number of 100 Character String Entries",
↪"plots": [], "x_label": "Number of Keys"}
```

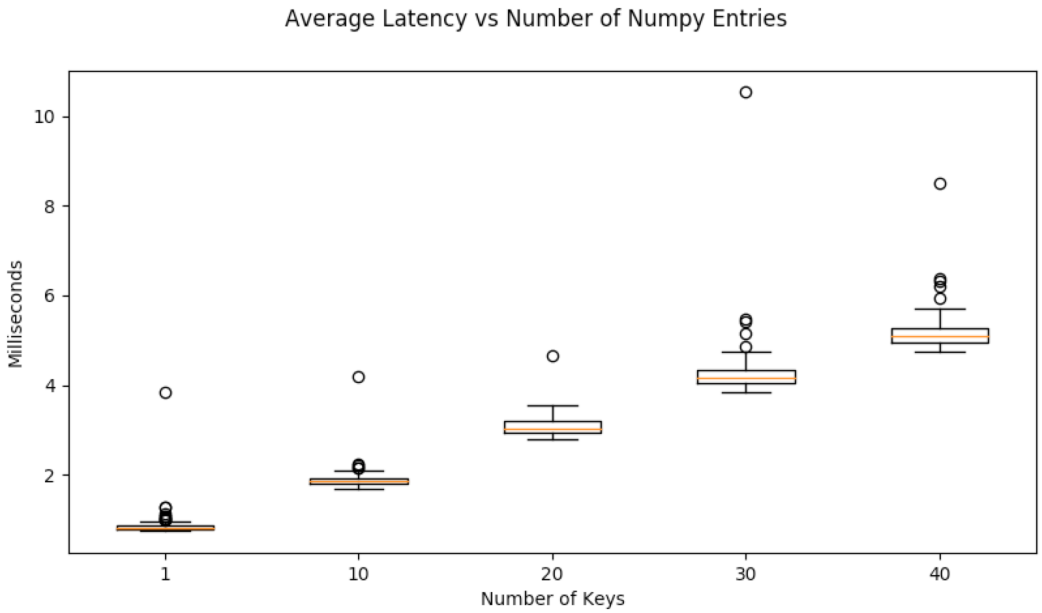
(continues on next page)

(continued from previous page)

```
    for copies in [max(1, 100 * i) for i in range(5)]:
        data = single_level_dictionary(copies=copies, data={"single_key": "".join(["A
↪" for i in range(10**2)])})
        p = {
            "ticker_label": copies,
            "times": multitrial_time_test(set, {"keys": ["key_growth"], "value": data}
↪, iterations=100)
        }
        info["plots"].append(p)
        print("Completed: {}".format(copies))
    plot_performance(info)
```

Numpy Data

In the below test, a dictionary with X numpy array entries was written to Redis. One hundred trials were conducted. The box-plots of their latency is below. Latency grows linearly with the amount of data sent. The source code to generate this plot is below.

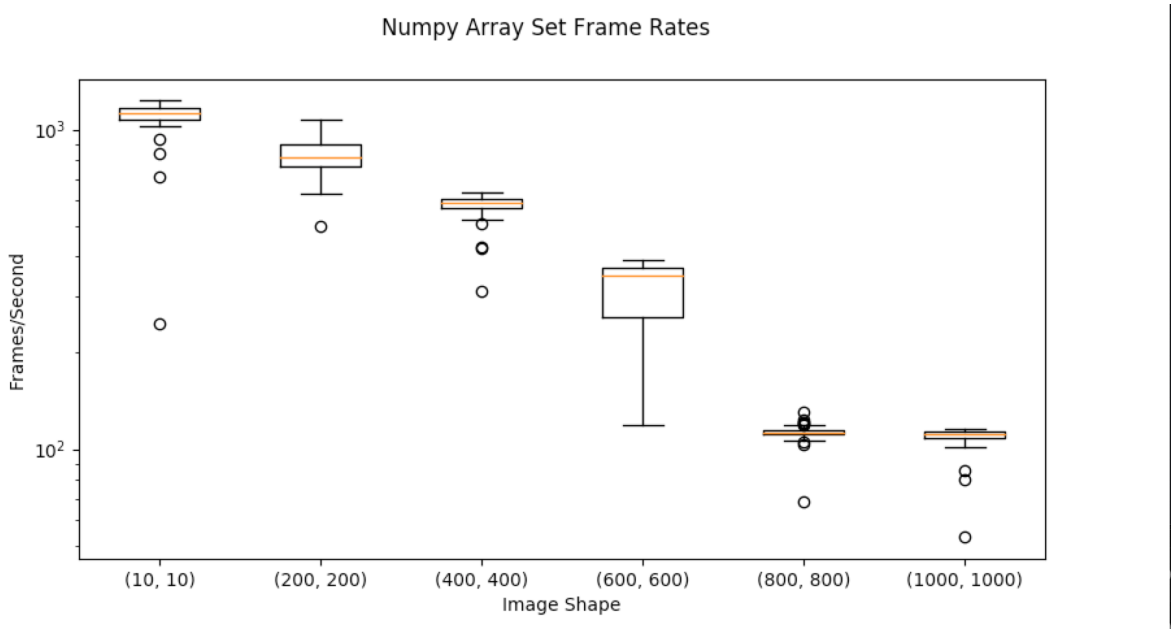



```
def key_growth_numpy():
    info = {"title": "Average Latency vs Number of Numpy Entries", "plots": [], "x_
↪label": "Number of Keys"}
    for copies in [max(1, 10 * i) for i in range(5)]:
        data = single_level_dictionary(copies=copies, data={"single_key": np.random.
↪rand(3, 4)})
        p = {
            "ticker_label": copies,
            "times": multitrial_time_test(set, {"keys": ["key_growth_numpy"], "value
↪": data}, iterations=100)
        }
        info["plots"].append(p)
        print("Completed: {}".format(copies))
    plot_performance(info)
```

5.1.2 Numpy Array Size Throughput

Set

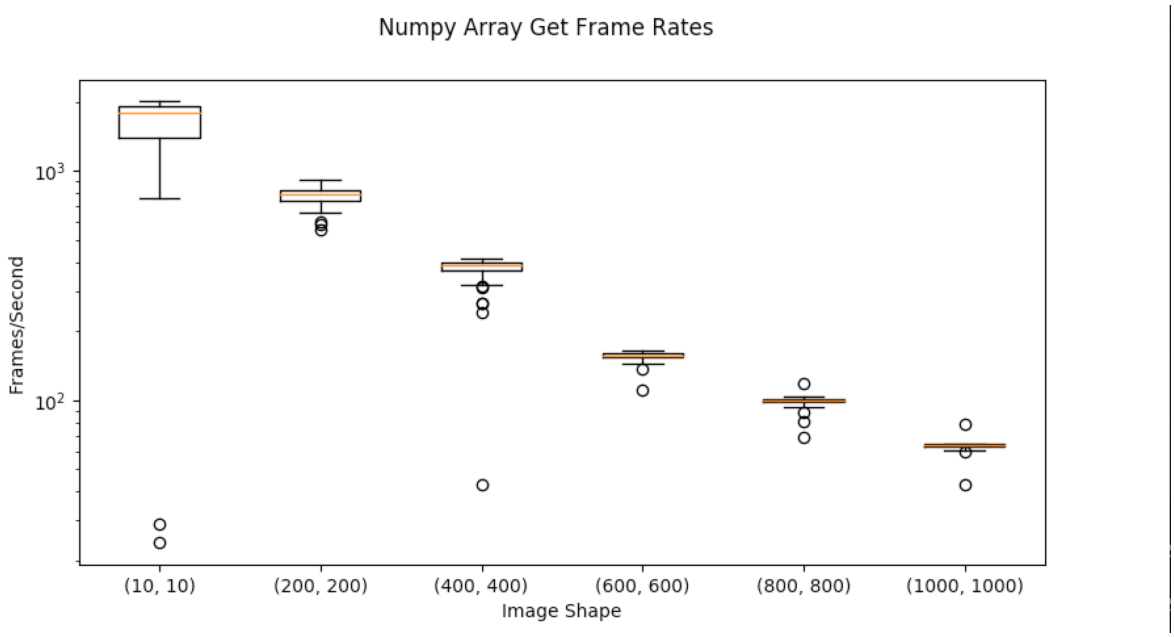
A numpy array of size (N x N) was uploaded to the server as fast as possible. Frame rates are shown below




```
def numpy_set_frame_rates():
    info = {"title": "Numpy Array Set Frame Rates", "plots": [], "x_label": "Image␣
↪Shape", "y_label": "Frames/Second", "y_scale":'log'}
    # sets = [np.random.rand(640, 480, 3), np.random.rand(720, 480, 3), np.random.
↪rand(1080, 720, 3)]
    sets = [np.random.rand(max(10, 200 * i), max(10, 200 * i)) for i in range(6)]
    for arr in sets:
        trials = multitrial_time_test(set, {"keys": ["np_frame_rate_test", "key"],
↪"value": arr}, iterations=50)
        trials = [1000.0/t for t in trials]
        p = {
            "ticker_label": arr.shape,
            "times": trials
        }
        info["plots"].append(p)
    plot_performance(info)
```

Get

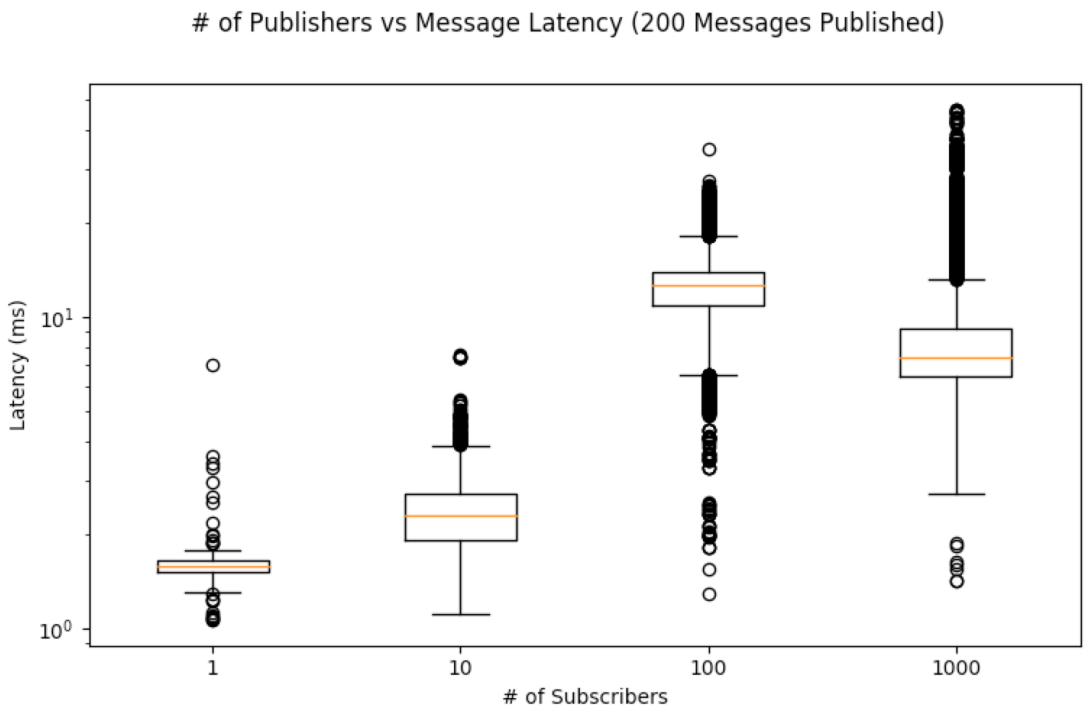
A numpy array of size (N x N) was downloaded from the server as fast as possible. Frame rates are shown below




```
def numpy_get_frame_rates():
    info = {"title": "Numpy Array Get Frame Rates", "plots": [], "x_label": "Image␣
↪Shape", "y_label": "Frames/Second", "y_scale":'log'}
    # sets = [np.random.rand(640, 480, 3), np.random.rand(720, 480, 3), np.random.
↪rand(1080, 720, 3)]
    sets = [np.random.rand(max(10, 200 * i), max(10, 200 * i)) for i in range(6)]
    for arr in sets:
        kvs["read_frame_rate_test"]["subkey"] = arr
        trials = multitrial_time_test(get, {"keys": ["read_frame_rate_test", "subkey
↪"]}, iterations=50)
        trials = [1000.0 / t for t in trials]
        p = {
            "ticker_label": arr.shape,
            "times": trials
        }
```

5.2 Subscriber Overhead

The below code tested what the overhead was with having multiple subscribers to a given channel. The publisher and each subscriber was run in it's own process on the same machine. A publisher uploaded a timestamp and subscribers calculated the difference between the timestamp and the time they read the image.




```
# ---------------------------- Subscriber Overhead Testing ----------------------------


PULSE_GAP = 0.02
TRIALS = 200


def append_time_to_list(data, updated_path, times):
    times.append(time.time() - data["timestamp"])


def overhead_testing_subscriber(test_name, timeout=10):
    times = []
    interface = RedisInterface()
    subscriber = CallbackSubscriber("overhead_test", interface, append_time_to_list, {
↪"times": times})
    subscriber.listen()
    time.sleep(timeout)
    base = os.path.dirname(os.path.abspath(__file__))
    save_dir = os.path.join(base, "logs", "overhead_test", test_name)
    save_path = os.path.join(save_dir, "subscriber_{}.txt".format(os.getpid()))
    if not os.path.exists(save_dir):
        os.makedirs(save_dir)
    with open(save_path, "w") as f:
        for t in times:
            f.write("{}\n".format(t))


def overhead_testing_publisher():
    interface = RedisInterface()
    publisher = PublishSpace(interface)
    for i in range(TRIALS):
```

(continues on next page)

(continued from previous page)

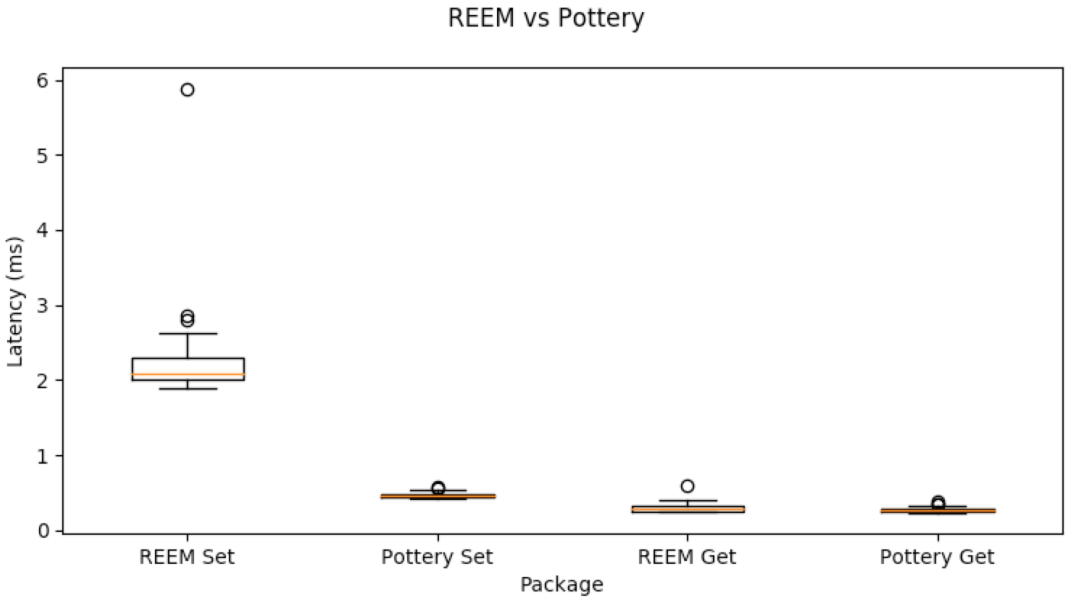
```
        publisher["overhead_test"] = {"timestamp": time.time()}
        time.sleep(PULSE_GAP)


def generate_subscriber_overhead_data(num_subscriber_list):
    base = os.path.join(os.path.dirname(os.path.abspath(__file__)), "logs", "overhead_
↪test")
    shutil.rmtree(base)
    for num_subscribers in num_subscriber_list:
        processes = [(overhead_testing_publisher, (), {})]
        test_name = "subs={}".format(num_subscribers)
        save_dir = os.path.join(base, test_name)
        if not os.path.exists(save_dir):
            os.makedirs(save_dir)

        for i in range(num_subscribers):
            processes.append((overhead_testing_subscriber, (test_name, PULSE_GAP *␣
↪TRIALS), {}))

        run_as_processes(processes)
        print("Completed test with {} subscribers".format(num_subscribers))


def plot_overhead_data():
    time_data = {}
    base = os.path.dirname(os.path.abspath(__file__))
    save_dir = os.path.join(base, "logs", "overhead_test")
    for dirpath, dirs, files in os.walk(save_dir):
        if "subs" not in dirpath:
            continue
        num_subscribers = int(dirpath.split("subs=")[1])
        time_data[num_subscribers] = []
        for fpath in files:
            with open(os.path.join(dirpath, fpath), 'r') as file:
                for line in file:
                    time_data[num_subscribers].append(float(line) * 1000)    # Seconds␣
↪to milliseconds conversion
    plot_info = {
        "title": "# of Publishers vs Message Latency ({} Messages Published)".
↪format(TRIALS),
        "x_label": "# of Subscribers",
        "y_label": "Latency (ms)",
        "y_scale": "log",
        "plots": []
    }
    for key, value in sorted(time_data.items(), key=lambda kv: kv[0]):
        plot_info["plots"].append({"ticker_label": key, "times": value})
    plot_performance(plot_info)


def overhead_tests_main():
    generate_subscriber_overhead_data([1, 10, 100, 1000])
    plot_overhead_data()
```

5.3 Comparison

There exist other packages that provide similar but not identical functionality to REEM.

5.3.1 Potteryx

The package potteryx offers pythonic ways of using Redis data types. It's implementation for setting JSON-compatible data is faster than REEM but gets are comparable.



The comparison was generated with the following code:

```
def set_pottery(redis_dict, value):
    redis_dict["data"] = value


def get_pottery(redis_dict, keys):
    ret = redis_dict
    for k in keys:
        ret = ret[k]
    return ret


def compare_to_potteryx():
    client = Redis.from_url('redis://localhost:6379/')
    pottery_dict = RedisDict(redis=client, key='pottery')

    info = {"title": "REEM vs Pottery", "plots": [],
            "x_label": "Package",
            "y_label": "Latency (ms)"}

    data = nested_level_dictionary(
        levels=5,
        data=single_level_dictionary(
```

(continues on next page)

(continued from previous page)

```
            copies=100,
            data={
                "single_key": "".join(["A" for i in range(10 ** 2)]),
                "nested_data": {
                    "subkey": "".join(["A" for i in range(10 ** 2)])
                }
            }
        )
    )
    # REEM Set
    p = {
        "ticker_label": "REEM Set",
        "times": multitrial_time_test(set, {"keys": ["pottery_comparison"], "value":␣
↪data}, iterations=100)
    }
    info["plots"].append(p)

    # Pottery Set
    p = {
        "ticker_label": "Pottery Set",
        "times": multitrial_time_test(set_pottery, {"redis_dict": pottery_dict, "value
↪": data}, iterations=100)
    }
    info["plots"].append(p)

    reem_read_path = path_to_key_sequence(".pottery_comparison.sub_0.sub_1.sub_2.sub_
↪3.sub_4.copy_0_single_key")
    pottery_read_path = path_to_key_sequence(".data.sub_0.sub_1.sub_2.sub_3.sub_4.
↪copy_0_single_key")
    # REEM Get
    p = {
        "ticker_label": "REEM Get",
        "times": multitrial_time_test(get, {"keys": reem_read_path}, iterations=100)
    }
    info["plots"].append(p)

    # Pottery Get
    p = {
        "ticker_label": "Pottery Get",
        "times": multitrial_time_test(get_pottery, {"redis_dict": pottery_dict, "keys
↪": pottery_read_path}, iterations=100)
    }
    info["plots"].append(p)

    plot_performance(info)
```

Pottery does not use ReJSON. To store nested data, `potteryx` serializes data deeper than one level inside a Python dictionary to JSON. The JSON is then stored as a subkey of Redis Hash.

Pottery will require some extra work to get non-serializable data like numpy arrays to work with it.

CHAPTER

SIX

SERVER UTILITIES

REEM comes with some server-side utilities to help make debugging a little bit easier. They are located together inside a GitHub repository. It is not a PyPi package. Download the repository and install dependencies with the below script

```
git clone git@github.com:tn74/reem-server.git
cd reem-server
pip3 install -r requirements.txt
```

6.1 Browser

The browser allows you to view data inside a Redis server in a web format. It is written with the python Django web framework. Make sure you have a Redis server turned on before starting it. To start the browser's server, run the following script starting at the `reem-server` directory

```
cd browser
python manage.py runserver
```

Some Notes:

1. The web-browser-server assumes that the Redis server is on the same machine and bound to the localhost interface. If the browser is being launched from a different computer as the Redis server, change the REEM_HOSTNAME variable at the bottom of the django configuration file

2. Running `python manage.py runserver` starts the server on the localhost interface. If you want to connect to this web browser from other machines, run `python manage.py runserver 0.0.0.0:8000` and access the browser at the ip address of the machine running the browser and port 8000.

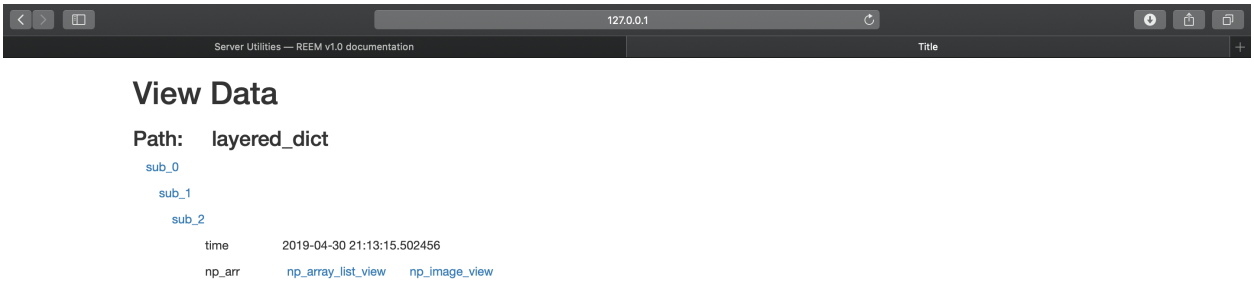
3. To access data at a specific path in redis, go to `http://localhost:8000/view/<reem-path>`. For example, if you wanted to see what was stored at "foo.bar.subkey", go to the url "http://127.0.0.1:8000/view/foo.bar.subkey"

4. Numpy data can be viewed in two ways

    1. A pretty printed list of numbers
    2. An image

    - If you try to view an image-sized numpy array as a list of pretty printed numbers, the server will be very slow.

A screen capture of what the browser looks like is below



6.2 Logger

We want to implement logging functionality that ultimately allows users to see how specific keys change in Redis over time. Imagine having the above browser with a slider bar that allows you to see how a key changes as you drag the slider. We have begun testing two ways of doing this but neither is fully functional.

6.2.1 RDB Logger

Redis has two natural ways of storing data to persistent memory. It can use RDB files that snapshot the database at a specific point in time and AOF files that track all changes to Redis in an append only fashion.

RDB

Redis can be told to save the database to an RDB file periodically but it is configured to always write to the same file. This poses a problem if we would like to save the state of data at previous points in time.

There is a script in the reem-server repository that copies the redis's dump file periodically to a folder so users can save snapshots of Redis data in time. The script is called according to the syntax

```
python reem-logger <path-to-directory-of-snapshots> <path-to-redis-dump-file>
<seconds-between-snapshot>
```

The next step is to select a snapshot based on a timestamp and load a Redis server with it. When Redis is executed inside a directory, Redis checks that directory for a file called `dump.rdb` and automatically loads it. If the file does not exist, Redis will create it. To restore data from a particular rdb file, copy that rdb-file to the directory in which you will run Redis and rename the rdb-file to `dump.rdb`. We have not provided functionality for this in python because it is a simple copy, paste, and rename that depends mainly on your paths.

There are some existing tools that allow the user to parse through RDB directly without starting a Redis server, but they generally do not support parsing ReJSON commands since ReJSON is a young third-party module.

AOF

Ideally we would not have to copy data that doesn't change much like we do when we save so many RDB files. We would like to be able to use the AOF file that tracks all changes made to the Redis server. It is played back by a Redis server when it is used to restore a specific state. More research must be done into finding parsers for AOF files.

6.2.2 Custom Logger

Some work was done on developing a custom logger. This custom program would not use a standard Redis data saving format but would use REEM to retrieve data from Redis periodically and use numpy to store it. The user would be able to specify a particular frequency for a given key. The code is online here

This log function would take in a key file that specified a paths and periods (representing how frequently to read a specific path in Redis) and an output directory to store saved data.

CHAPTER

SEVEN

DOCS

class reem.datatypes.**Writer**(*top_key_name*, *interface*)

Responsible for setting data inside Redis

The Writer class is an internal class that is used for all data sent to Redis (not including pub/sub messages). Each key that will have nested data below requires a new instantiation of Writer

top_key_name

The name of the Redis key under which JSON data will be stored. To Redis, this will become a ReJSON key name. It is also used to generate the Redis key name that `ship`'s use to store non JSON data.

Type str

interface

Defines the connection to Redis this writer will use

Type RedisInterface

send_to_redis(*set_path*, *set_value*)

Execute equivalent of `JSON.SET self.top_key_name <set_path> <set_value>`

From the user's perspective, it executes `JSON.SET self.top_key_name <set_path> <set_value>` except that `set_value` can be json-incompatible. This is the only public method of this class. It determines what non-serializable types are inside set_value, stores the serializable data as a JSON, and stores the non-serializable data using `self.interface`'s ships

Parameters

- **set_path** (*str*) – path underneath JSON key to set
- **set_value** – value to set

Returns None

_Writer__initialize_metadata()

Pull metadata for this key from Redis or set a default

Returns None

_Writer__process_metadata(*set_path*, *set_value*)

Handle metadata updates

Given the path and value the user would like to set, check if there are non-serializable data types and update metadata locally and in Redis. Happens without pipeline

Parameters

- **set_path** (*str*) – path underneath JSON key to set
- **set_value** – value to set

Returns None

_Writer__publish_non_serializables(*set_path*, *set_value*)
Publish JSON incompatible data to Redis

Given a set, publish the non-serializable components to redis, given that metadata has been updated already

Parameters

- **set_path** (*str*) – path underneath JSON key to set
- **set_value** – value to set

Returns None

_Writer__publish_serializables(*set_path*, *set_value*)
Publish the serializable portion of ‘‘set_value

Take out the non-serializable part of set_value and publish it at set_path

Parameters

- **set_path** (*str*) – path underneath JSON key to set
- **set_value** – value to set

Returns: None

class reem.datatypes.**Reader**(*top_key_name*, *interface*)
Responsible for getting data from Redis

The Reader class is an internal class that is used for all read from Redis (not including pub/sub messages). Each key that will have nested data below requires a new instantiation of Reader

top_key_name
The name of the Redis key under which JSON data is stored

Type str

interface
Defines the connection to Redis this reader will use

Type RedisInterface

read_from_redis(*read_path*)
Read specified path from Redis

This is the only public method of the Reader class. It will retrieve the data stored at a specified path from Redis. At a high level, it reads data stored with ReJSON and inserts non-JSON compatible data at appropriate paths using the metadata associated with this key.

Parameters **read_path** (*str*) – path the user wants to read

Returns: data stored at value in Redis

update_metadata()
Update the local copy of metadata if a relevant path has been updated.

The metadata listener is a redis client subscribed to key-space notifications. If a relevant path is updated, this Reader’s `pull_metadata` flag will be turned on. If `pull_metadata` is `True`, then the reader will fetch metadata from the Redis server.

Returns: None

queue_reads(*read_path*)
Queue reads in a pipeline

Queue all redis queries necessary to read data at path into the appropriate redis pipeline. First, queue decoded pipeline with the ReJSON query Next, queue all the special path reads with the non-decoded pipeline and ships

Parameters **read_path** – path user wants to read

Returns: None

build_dictionary(*read_path*)
Execute pipelines and consolidate data into a dictionary

Parameters **read_path** – path user wants to read

Returns: The data stored at `read_path` in Redis

pull_special_path(*read_path*)
Directly pull a non-JSON path

If the user specified path is not in JSON, this will retrieve the data directly without going through ReJSON.

Parameters **read_path** – path user wants to read

Returns:

class reem.datatypes.**KeyValueStore**(*interface*)
Dictionary Like object used for get/set paradigm

The `KeyValueStore` object is one that users will frequently use. It keeps `Reader` and `Writer` objects for each key that the user read or written to. It does not actually handle the getting and setting of data but produces `ReadablePathHandler` objects that assist with path construction and call the reader and writer's write and read methods.

interface
Defines the connection to Redis this reader will use

Type RedisInterface

track_schema_changes(*set_value*, *keys=None*)
Performance optimization for skipping schema update checks

Stop checking for schema updates when setting data. Use ONLY if your data's schema is static and you are really trying to eek out every bit of optimization.

Parameters

- **set_value** (*bool*) – True/False indicating if the keys' schema should be tracked
- **keys** (*List[str]*) – List of keys to track. If None, all present and future keys are tracked
- **to set_value** (*according*) –

Returns: None

_KeyValueStore__ensure_key_existence(*key*)
Ensure that the requested key has a reader and writer associated with it.

Returns: None

class reem.datatypes.**Publisher**(*top_key_name*, *interface*)
Defines Publisher behavior

The Publisher is identical to the writer but publishes a message when it writes a value.

send_to_redis(*set_path*, *set_value*)
Publisher equivalent of Writer `send_to_redis`

This is an equivalent function to `Writer`'s `send_to_redis` method but also publishes a message indicating what channel has been updated.

Parameters

- **set_path** (*str*) – path underneath JSON key to set
- **set_value** – value to set

Returns: None

_Writer__initialize_metadata()
Pull metadata for this key from Redis or set a default

Returns None

_Writer__process_metadata(*set_path*, *set_value*)
Handle metadata updates

Given the path and value the user would like to set, check if there are non-serializable data types and update metadata locally and in Redis. Happens without pipeline

Parameters

- **set_path** (*str*) – path underneath JSON key to set
- **set_value** – value to set

Returns None

_Writer__publish_non_serializables(*set_path*, *set_value*)
Publish JSON incompatible data to Redis

Given a set, publish the non-serializable components to redis, given that metadata has been updated already

Parameters

- **set_path** (*str*) – path underneath JSON key to set
- **set_value** – value to set

Returns None

_Writer__publish_serializables(*set_path*, *set_value*)
Publish the serializable portion of ‘‘set_value

Take out the non-serializable part of set_value and publish it at set_path

Parameters

- **set_path** (*str*) – path underneath JSON key to set
- **set_value** – value to set

Returns: None

class reem.datatypes.**PublishSpace**(*interface*)
Convenience class for publishing

This class keeps track of `Publisher` objects for each key the user has published on.

_KeyValueStore__ensure_key_existence(*key*)
Ensure that the requested key has a reader and writer associated with it.

Returns: None

_PublishSpace__ensure_key_existence(*key*)
Identical to `KeyValueStore` method of same name but does not instantiate a `Reader` object

track_schema_changes(*set_value*, *keys=None*)
Performance optimization for skipping schema update checks

Stop checking for schema updates when setting data. Use ONLY if your data's schema is static and you are really trying to eek out every bit of optimization.

Parameters

- **set_value** (*bool*) – True/False indicating if the keys' schema should be tracked
- **keys** (*List[str]*) – List of keys to track. If None, all present and future keys are tracked
- **to set_value** (*according*) –

Returns: None

class reem.datatypes.**SilentSubscriber**(*channel*, *interface*)
Silent Subscriber Implementation

update_local_copy(*channel*, *message*)
Update the local copy of the data stored under this channel name in redis.

Parameters

- **channel** – the name of the channel that was published.
- **message** – message published on that channel

Returns: None

listen()
Makes this subscriber start listening

Returns: None

value()
Get data stored at root

Where as the reader can do `server["foo"].read()` with if server is a `KeyValueStore`, accessing the root value of a subscriber is not as easy. This method retrieves all data stored underneath a top level key.

Returns: all data stored underneath a top level Redis key.

build_dictionary(*read_path*)
Execute pipelines and consolidate data into a dictionary

Parameters **read_path** – path user wants to read

Returns: The data stored at `read_path` in Redis

pull_special_path(*read_path*)
Directly pull a non-JSON path

If the user specified path is not in JSON, this will retrieve the data directly without going through ReJSON.

Parameters **read_path** – path user wants to read

Returns:

queue_reads(*read_path*)
Queue reads in a pipeline

Queue all redis queries necessary to read data at path into the appropriate redis pipeline. First, queue decoded pipeline with the ReJSON query Next, queue all the special path reads with the non-decoded pipeline and ships

Parameters **read_path** – path user wants to read

Returns: None

read_from_redis(*read_path*)
Read specified path from Redis

This is the only public method of the Reader class. It will retrieve the data stored at a specified path from Redis. At a high level, it reads data stored with ReJSON and inserts non-JSON compatible data at appropriate paths using the metadata associated with this key.

Parameters **read_path** (*str*) – path the user wants to read

Returns: data stored at value in Redis

update_metadata()
Update the local copy of metadata if a relevant path has been updated.

The metadata listener is a redis client subscribed to key-space notifications. If a relevant path is updated, this Reader's `pull_metadata` flag will be turned on. If `pull_metadata` is `True`, then the reader will fetch metadata from the Redis server.

Returns: None

class reem.datatypes.**CallbackSubscriber**(*channel*, *interface*, *callback_function*, *kwargs*)
Callback Subscriber Implementation

call_user_function(*channel*, *message*)
Wrapper callback function (wrapping user function) for this class to work with a RawSubscriber object Fits required interface for a ChannelSubscriber callback function :param channel: channel published to :param message: message that was published :return: None :rtype: None

build_dictionary(*read_path*)
Execute pipelines and consolidate data into a dictionary

Parameters **read_path** – path user wants to read

Returns: The data stored at `read_path` in Redis

listen()
Makes this subscriber start listening

Returns: None

pull_special_path(*read_path*)
Directly pull a non-JSON path

If the user specified path is not in JSON, this will retrieve the data directly without going through ReJSON.

Parameters **read_path** – path user wants to read

Returns:

queue_reads(*read_path*)
Queue reads in a pipeline

Queue all redis queries necessary to read data at path into the appropriate redis pipeline. First, queue decoded pipeline with the ReJSON query Next, queue all the special path reads with the non-decoded pipeline and ships

Parameters **read_path** – path user wants to read

Returns: None

read_from_redis(*read_path*)
Read specified path from Redis

This is the only public method of the Reader class. It will retrieve the data stored at a specified path from Redis. At a high level, it reads data stored with ReJSON and inserts non-JSON compatible data at appropriate paths using the metadata associated with this key.

Parameters **read_path** (*str*) – path the user wants to read

Returns: data stored at value in Redis

update_local_copy(*channel*, *message*)
Update the local copy of the data stored under this channel name in redis.

Parameters

- **channel** – the name of the channel that was published.
- **message** – message published on that channel

Returns: None

update_metadata()
Update the local copy of metadata if a relevant path has been updated.

The metadata listener is a redis client subscribed to key-space notifications. If a relevant path is updated, this Reader's `pull_metadata` flag will be turned on. If `pull_metadata` is `True`, then the reader will fetch metadata from the Redis server.

Returns: None

value()
Get data stored at root

Where as the reader can do `server["foo"].read()` with if server is a `KeyValueStore`, accessing the root value of a subscriber is not as easy. This method retrieves all data stored underneath a top level key.

Returns: all data stored underneath a top level Redis key.

CHAPTER

EIGHT

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

r

INDEX

Symbols

B

C

G

I

K

L

P

Q

R

S

T

U

V

W