# Benchmarks of `redis-py`, `hiredis-py` and string creation

@mokaddem

March 5, 2018

# Contents

# Introduction

In the course of a software development, I needed a buffering medium which was able to fulfil these requirements:

- Capable of handling millions of request per second
- Data persistence
- Easy to use

Where Redis clearly suits them.

My software being written in python, I evidently used one of the redis python client available; `redis-py` is the recommended one.

This report focuses only on throughput, so data size was fixed and was fitting my needs.
It contains benchmarks of `redis-py`, where we can see that it is particularly slow compared to the c-written `hiredis` library or `redis-benchmark`. It also presents some features and techniques to boost performances, as well as comparing methods to rapidly create strings.

# Chapter 1

# Aggregated results

## 1.1 Summary

These tables and plots show a summary for each tested different configuration.

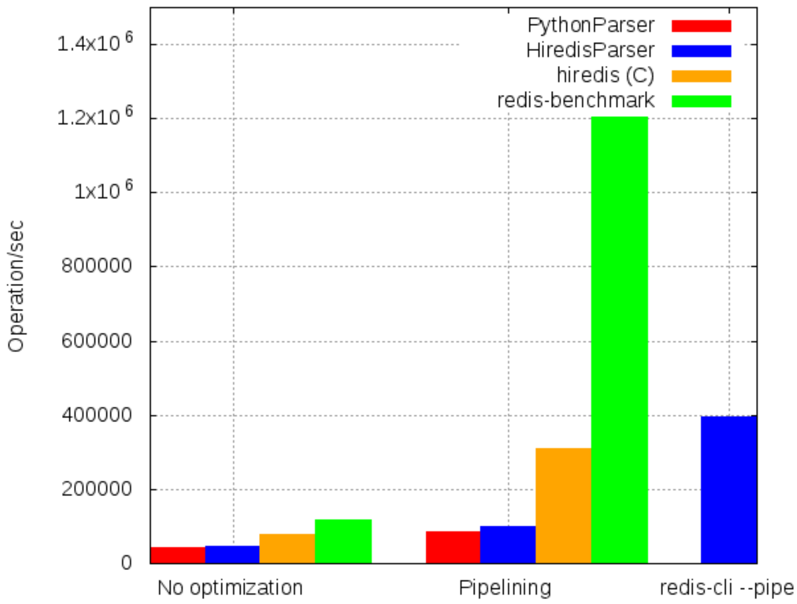
### 1.1.1 Pushing - `LPUSH`

| | Tool/feature | Parser | Total time taken (s) | | | Operation/sec | | |
|---|---|---|---|---|---|---|---|---|
| | | | worst | best | average | worst | best | average |
| 2.2 | Naive | PythonParser | 2.86 | 2.20 | 2.37 | 34,942 | 45,464 | 42,158 |
| | Naive | HiredisParser | 2.20 | 2.09 | 2.17 | 45,388 | 47,745 | 46,150 |
| | Naive | hiredis (C)[1] | 1.33 | 1.23 | 1.25 | 75,363 | 81,365 | 79,977 |
| 2.3 | Pipelining `-P 1000` | PythonParser | 1.18 | 1.16 | 1.17 | 85,078 | 85,979 | 85,642 |
| | Pipelining `-P 1000` | HiredisParser | 1.01 | 0.96 | 0.98 | 98,863 | 104,368 | 101,535 |
| | Pipelining `-P 1000` | hiredis (C) | 0.35 | 0.31 | 0.32 | 281,466 | 320,858 | 308,386 |
| 2.5, 2.6.1 | `redis-cli`, `format` | HiredisParser | 0.28 | 0.23 | 0.25 | 358,529 | 437,094 | 394,046 |
| 2.6.2 | `redis-cli`, `concatenation +` | HiredisParser | 0.25 | 0.21 | 0.22 | 400,002 | 487,566 | 454,178 |
| 2.6.3 | `redis-cli`, `substitution %` | HiredisParser | 0.16 | 0.15 | 0.16 | 610,237 | 687,978 | 638,709 |
| - | `redis-cli`, (no generation) | HiredisParser | 0.08 | 0.07 | 0.07 | 1,287,382 | 1,523,811 | 1,400,926 |
| | `redis-benchmark -c 1 -d 129` | | | - | | | - | 117,508 |
| | `redis-benchmark -c 1 -d 129 -P 1000` | | | - | | | - | 1,204,819 |

### 1.1.2 Popping - `RPOP` or `LRANGE/LTRIM`

| | Tool/feature | parser | Total time taken (s) | | | Operation/sec | | |
|---|---|---|---|---|---|---|---|---|
| | | | worst | best | average | worst | best | average |
| 2.2 | Naive | PythonParser | 2.68 | 2.11 | 2.27 | 37,301 | 47,323 | 44,021 |
| | Naive | HiredisParser | 1.96 | 1.85 | 1.91 | 51,050 | 53,950 | 52,383 |
| | Naive | hiredis (C) | 1.10 | 1.05 | 1.07 | 90,656 | 95,411 | 93,724 |
| 2.3 | Pipelining | PythonParser | 1.10 | 1.09 | 1.09 | 90,862 | 91,730 | 91,402 |
| | Pipelining | HiredisParser | 0.83 | 0.77 | 0.80 | 121,207 | 129,375 | 124,924 |
| | Pipelining | hiredis (C) | 0.15 | 0.13 | 0.14 | 659,078 | 751,359 | 720,310 |
| 2.4 | `LRANGE` trick | PythonParser | 0.34 | 0.32 | 0.33 | 292,466 | 311,161 | 304,744 |
| | `LRANGE` trick | HiredisParser | 0.08 | 0.07 | 0.07 | 1,207,644 | 1,511,085 | 1,416,695 |
| | `LRANGE` trick | hiredis (C) | 0.05 | 0.04 | 0.04 | 2,152,203 | 2,427,007 | 2,318,716 |
| | `redis-benchmark -c 1 -d 129 (RPOP)` | | | - | | | - | 116,822 |
| | `redis-benchmark -c 1 -d 129 -P 1000 (RPOP)` | | | - | | | - | 1,136,363 |
| | `redis-benchmark -c 1 -d 129 (LRANGE_100)` | | | - | | | - | 4,048,000 |

[1] Official C client



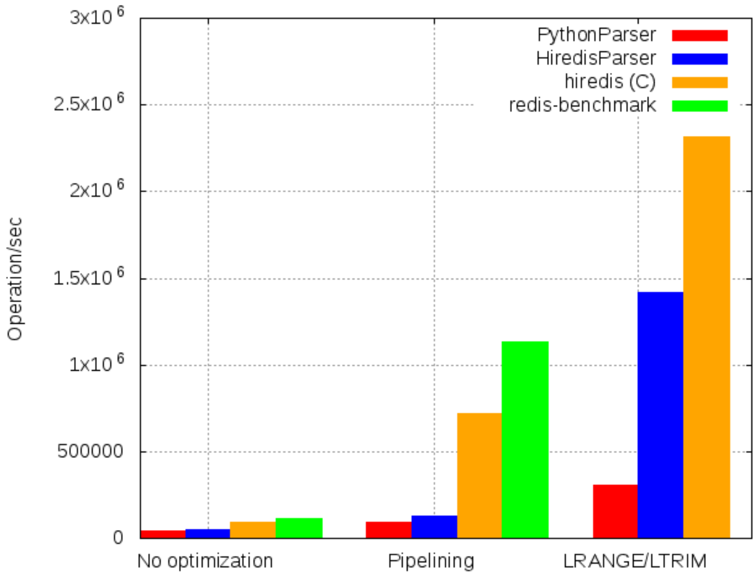

<u>`LPUSH:`</u>

With no optimization, python performs close to 60% of `hiredis (C)` performance:

$$\frac{\texttt{python perf}}{\texttt{hiredis (C) perf}} = \frac{46,150}{79,977} = 57.70\%$$

With pipelining, python performs close to only 30% of `hiredis (C)` performance:

$$\frac{\texttt{python perf}}{\texttt{hiredis (C) perf}} = \frac{101,535}{308,386} = 30.91\%$$



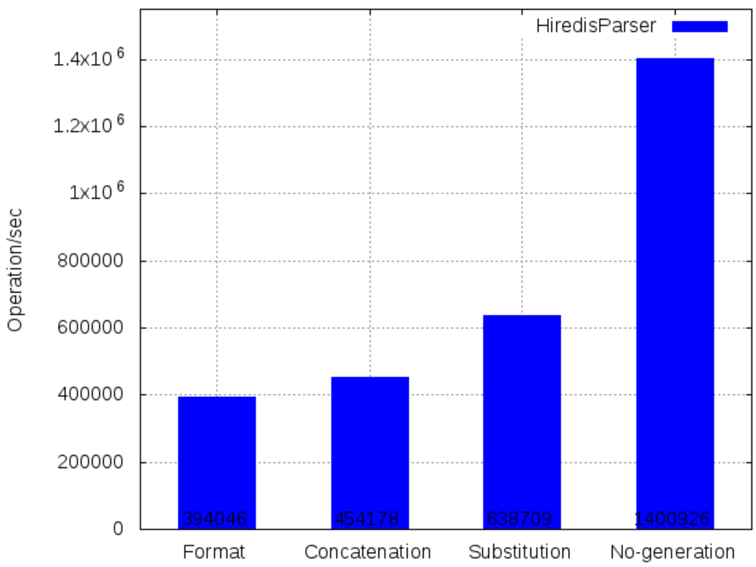

<u>`POP:`</u>

With no optimization, python performs close to 60% of `hiredis (C)` performance::

$$\frac{\texttt{python perf}}{\texttt{hiredis (C) perf}} = \frac{52,383}{93,724} = 55.89\%$$

With pipelining, python performs close to only 13% of `hiredis (C)` performance:

$$\frac{\texttt{python perf}}{\texttt{hiredis (C) perf}} = \frac{91,402}{720,310} = 12.69\%$$




<u>String Generation:</u>

| Method | Cummulative gain |
|---|---|
| `format` | 0% |
| `Concatenation` | 15.26% |
| `Substitution` | 40.63% |

# Chapter 2

# Detailed implentations and tests

## 2.1 Benchmark parameters

All benchmarks have been performed using these parameters

- Usage of redis `unix_socket_path`
- Payload: size = 129 bytes

```
'{"origin": null, "channel": 0, "content": "redis@tshark_save:53619abd-a27c-432c-8f8d
   -1d059aab5f24", "size": 54, "redirect": true}'
```

- Operations:
  - 100000 `LPUSH`
  - Number of `POP` may differ if we use `RPOP` or `LRANGE/LTRIM`
- For each tests, we are using two different parsers for responses: PythonParser and HiredisParser[1]
- In order to simulate a working software popping and adding elements to a buffer, the logic of the benchmark is the folloging:

```
# push
# c =   total count     = 100,000
# d =   divisor         = 1,000
# c/d = iteration count = 100
for i in range(int(c/d)):
    for j in range(d):
        # push
    for j in range(d):
        # pop
```

- For each benchmark, the processing has been done 10 times, then averaged
- For each 100 $\left(\frac{100{,}000}{1{,}000}\right)$ iterations, we are pushing and popping 1,000 elements

[1] Hiredis is a C library that is available with Python bindings, `redis-py` will attempt to use the HiredisParser if you have the hiredis module installed and will fallback to the PythonParser otherwise.

## 2.2 Naive implementation

For each payload to be buffered, we push them immediatly. Then, we retrieve them one at a time with a simple `POP` command.

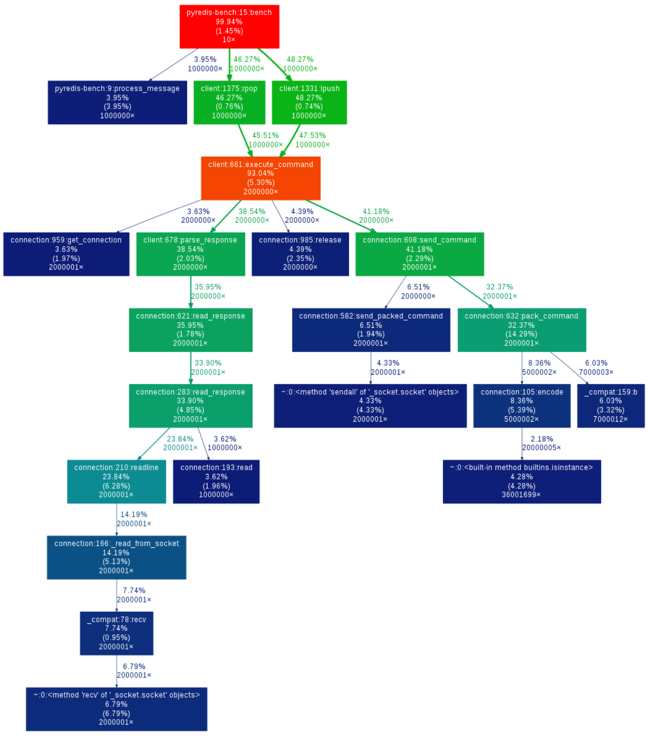
### Source code

```
# push
t1=time.time()
for j in range(d):
    redis.lpush('k', payload)
    cpush+=1
time_push += time.time()-t1
# pop
t1=time.time()
for j in range(d):
    msg = redis.rpop('k')
    process_message(msg)
    cpop+=1
time_pop += time.time()-t1
```

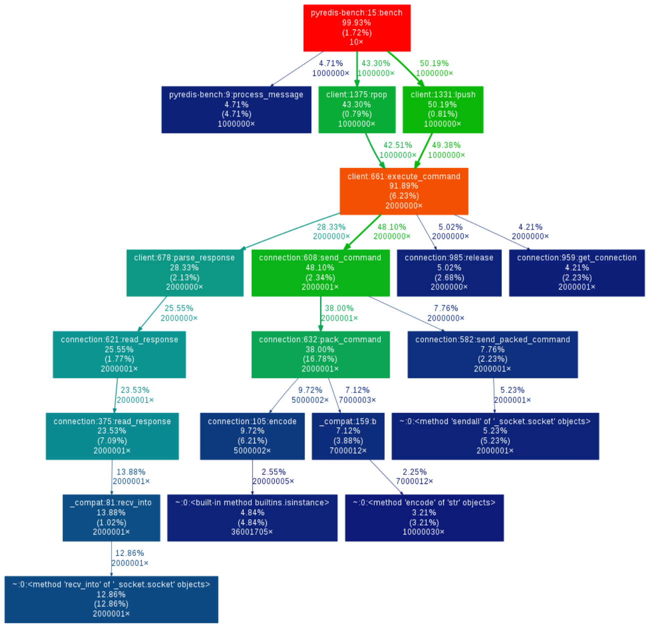
### Profiling summary

| Cumulative CPU % | PythonParser | HiredisParser |
|---|---|---|
| Processing | 3.95% | 4.71% |
| Receiving response | 38.54% | 28.33% |
| Sending command | 41.18% | 48.10% |
| Parser total gain: **19.24%** | | |

### PythonParser




### HiredisParser



## 2.3 Pipeline feature

Here, we use the redis pipeline[2] feature. For each payload to be buffered, we push them in a pipeline wich will execute pending commands every 1000 pushing operations. Then, we retrieve them by sending the POP command in a pipeline which will also execute pending commands every 1000 popping operations.

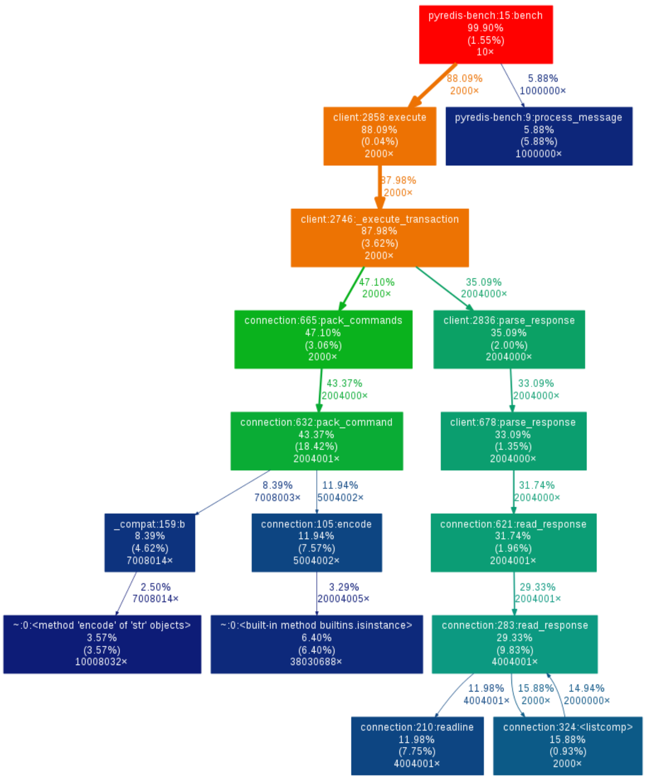
**Source code**

```
# push
t1=time.time()
for j in range(d):
    pipeline.lpush('k', payload)
    cpush+=1
pipeline.execute()
tpush += time.time()-t1
# pop
t1=time.time()
for j in range(d):
    pipeline.rpop('k')
resp = pipeline.execute()
for msg in resp:
    process_message(msg)
    cpop+=1
tpop += time.time()-t1
```

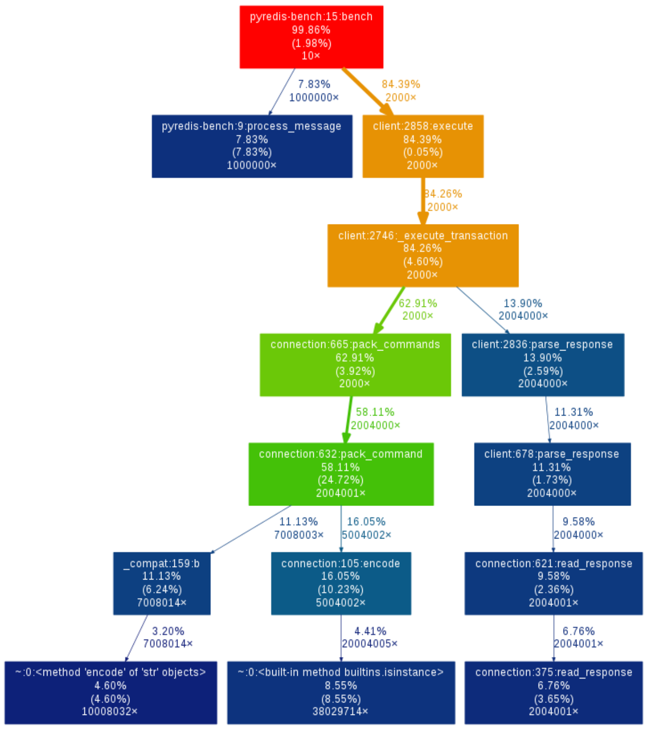
**Profiling summary**

| Cumulative CPU % | PythonParser | HiredisParser |
|---|---|---|
| Processing | 5.88% | 7.83% |
| Receiving response | 35.09% | 13.90% |
| Sending command | 47.10% | 62.91% |
| Parser total gain: | **33.16%** | |
| Gain compare to 2.2: | **48.86%** | **66.24%** |

**PythonParser**




**HiredisParser**




[2] Pipelining allows to send multiple commands to the server without waiting for the replies, and finally read the replies in a single step

## 2.4 Replacing `RPOP` by `LRANGE/LTRIM`

In this implementation, we are still using the redis pipeline feature for pushing data into the buffer, but we modify the popping behavior. Instead of sending one POP command at a time to the pipeline, we fetch a range (100 items) of buffered data with `LRANGE`, then we trim the buffer so that it mirror the effect of a `POP` command.

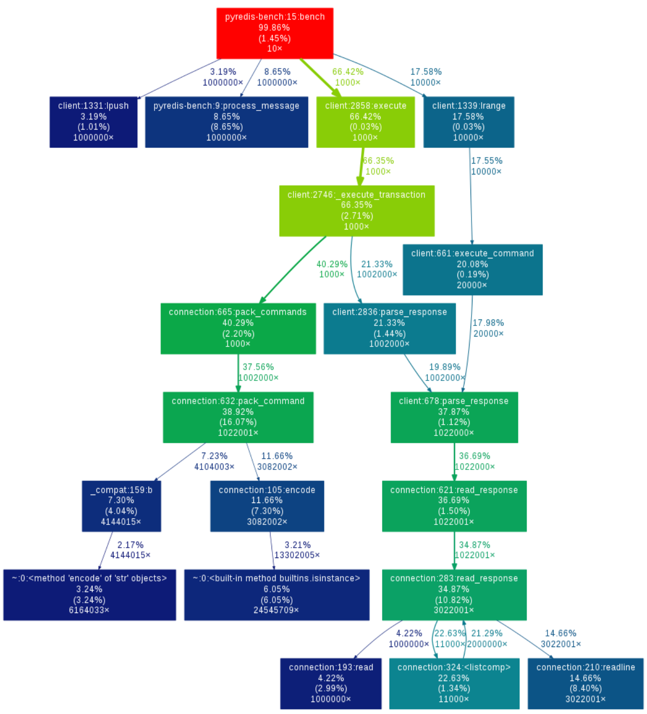
### Source code

```
# lrange_count = 100
# push
t1=time.time()
for j in range(d):
    pipeline.lpush('k', payload)
    cpush+=1
pipeline.execute()
tpush += time.time()-t1
# pop
t1=time.time()
for j in range(int(d/lrange_count)+1):
    msg_list = redis.lrange('k', -
   lrange_count, -1)
    redis.ltrim('k', 0, -len(msg_list)-1)
    for msg in msg_list:
        process_message(msg)
        cpop+=1
tpop += time.time()-t1
```

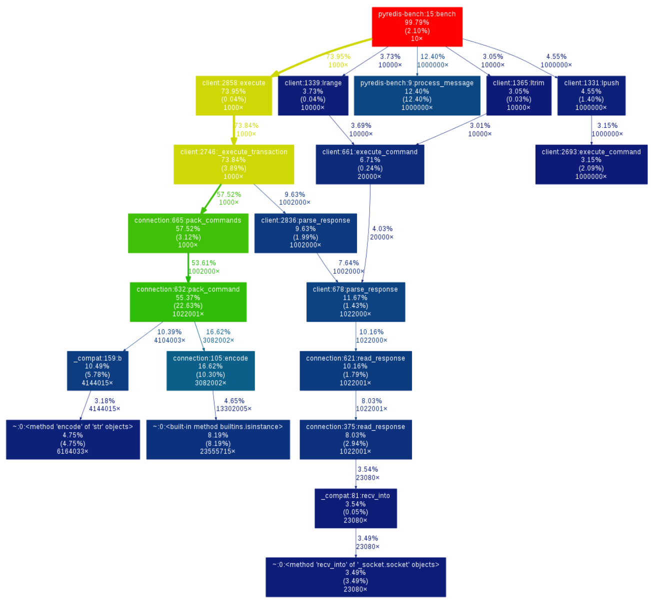
### Profiling summary

| Cumulative CPU % | PythonParser | HiredisParser |
|---|---|---|
| Processing | 8.65% | 12.40% |
| Receiving response | 37.87% | 11.67% |
| Sending command | 40.29% | 57.52% |
| Parser total gain: | **43.35%** | |
| Gain compare to 2.3: | **47.11%** | **58.37%** |

### PythonParser



### HiredisParser



## 2.5 Using `redis-cli --pipe`

In this implementation, we are still using the redis pipeline feature and the `LRANGE/LTRIM` replacement of `POP`. However, we will use the `redis-cli` binary (provided by redis) with the `--pipe` options[3].

In order to use this binary, we have to generate the valid redis protocol for the wanted command, which will be written to the STDIN of `redis-cli`.

**Source code**

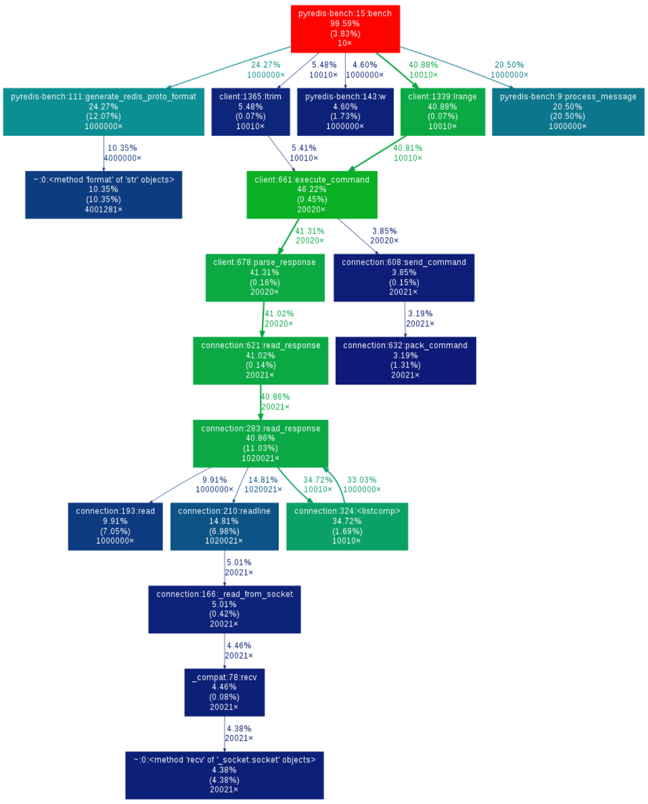
```python
# push with redis-cli
t1=time.time()
for j in range(d):
    write_to_stdin(generate_redis_protocol
    ('lpush', 'k', payload))
    cpush+=1
tpush += time.time()-t1
# pop with pipeline
t1=time.time()
for j in range(int(d/lrange_count)+1):
    msg_list = redis.lrange('k', -
    lrange_count, -1)
    redis.ltrim('k', 0, -len(msg_list)-1)
    for msg in msg_list:
        process_message(msg)
        cpop+=1
tpop += time.time()-t1
# flush and close stdin
# pop newly flushed items
```

**Profiling summary**

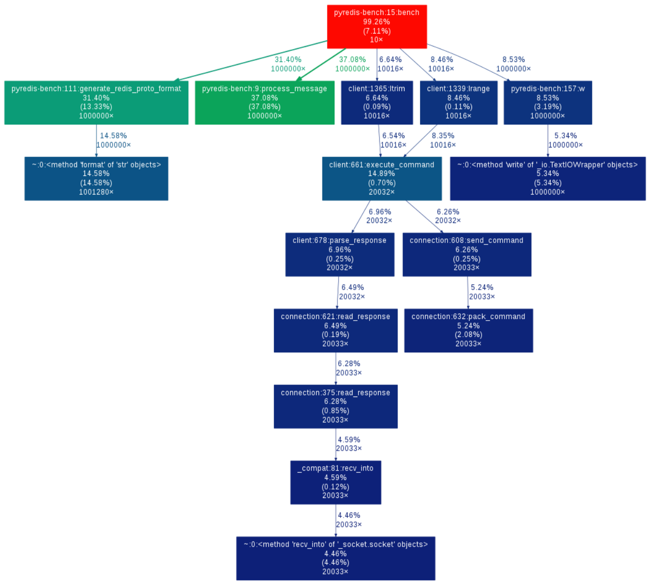
| Cumulative CPU % | PythonParser | HiredisParser |
|---|---|---|
| Processing | 20.50% | 37.08% |
| Receiving response | 41.51% | 6.96% |
| Sending command[a] | 8.45%[b] | 14.79%[c] |
| Parser total gain: | **80.87%** | |
| Gain compare to 2.4: | **136.99%** | **199.03%** |

[a]popping + mass insertion with `redis-cli`
[b]3.85% + 4.6%
[c]6.26% + 8.53%

**PythonParser**



**HiredisParser**



[3]`redis-cli --pipe` consists in writing new commands while you read replies at the same time, ignoring the round trip time for every command

## 2.6 Generating redis protocol

We saw in section 2.5 that we need to generate the redis protocol ourself. We will now see differents implentation of doing so, each one with drastically performance improvement.

For each algorythm, we use the same source code as in 2.5 with the `HiredisParser`.

### 2.6.1 Generating redis protocol with `string.Formatter`

```
def generate_redis_proto_format(cmd, key, value=''):
    cmd_split = cmd.split()
    if value == '':
        proto = '*{argNum}\r\n${argLen1}\r\n{arg1}\r\n${argLen2}\r\n{arg2}\r\n'.format(
                argNum=3 if value != '' else 2,
                argLen1=len(cmd), arg1=cmd,
                argLen2=len(key), arg2=key
                )
    else:
        proto = '*{argNum}\r\n${argLen1}\r\n{arg1}\r\n${argLen2}\r\n{arg2}\r\n${argLen3}\r\n{arg3}\r\n'.format(
                argNum=3 if value != '' else 2,
                argLen1=len(cmd), arg1=cmd,
                argLen2=len(key), arg2=key,
                argLen3=len(value), arg3=value)
    return proto
```

**Profiling summary**

It should be noted that the cumulative CPU time of `process_message` entirely depends on the function implentation. Still, it is given as a starting point for comparision.

| Cumulative CPU % | HiredisParser |
|---|---|
| `process_message` | 37.08% |
| `generate_redis_protocol_format` | 31.40% |
| Generation **loss**[4] | **46.87%** |

### 2.6.2 Generating redis protocol with `String` + concatenation operator

```
def generate_redis_proto_concat(cmd, key, value=''):
    cmd_split = cmd.split()
    proto = '*'+(str(3) if value != '' else str(2))+'\r\n'
    proto += '$'+str(len(cmd))+'\r\n'+cmd+'\r\n'
    proto += '$'+str(len(key))+'\r\n'+key+'\r\n'
    if value != '':
        proto += '$'+str(len(value))+'\r\n'+value+'\r\n'
    return proto
```

**Profiling summary**

| Cumulative CPU % | HiredisParser |
|---|---|
| `processing` | 40.53% |
| `generate_redis_protocol_string` | 26.34% |
| Gain compare to 2.6.1: | **9.30%** |
| Generation **loss** | **34.26%** |

[4] Cost of generating the protocol compated to not generating it a all.

### 2.6.3 Generating redis protocol with `String` % substitution operator

```
def generate_redis_proto_subst(cmd, key, value=''):
    cmd_split = cmd.split()

    if value != '':
        proto = '*%s\r\n$%s\r\n%s\r\n$%s\r\n%s\r\n$%s\r\n%s\r\n' % ((str(3) if value != '' else str(2)), len(cmd), cmd, len(key), key, len(value), value)
    else:
        proto = '*%s\r\n$%s\r\n%s\r\n$%s\r\n%s\r\n' % ((str(3) if value != '' else str(2)), len(cmd), cmd, len(key), key)
    return proto
```

#### Profiling summary

| Cumulative CPU % | HiredisParser |
|---|---|
| `processing` | 43.62% |
| `generate_redis_protocol_subst` | 19.92% |
| Gain compare to 2.6.2: | **7.62%** |
| Generation **loss** | **24.85%** |

# Conclusion

We saw that it is not recommended to use `redis-py` if we need pure throughput. Even so, some features like pipelining or using a different parser increase the performance reasonably, we are still far from those achieved by `hiredis (C)`.

In anycase, using the HiredisParser can increase performances up to 40% for our case. Don't forget: `redis-py` uses `HiredisParser` if it is installed, therefore:

```
pip install redis
pip install hiredis
```