Report for Message boards

1 Time to complete

The whole task took me about one and a half day. Half of the time used to get familiar with the interface of Redis (both redis shell and py-redis). The other half of the time used to consider about the system architecture, writing and debugging the code and write the report.

2 System Architecture

2.1 Overview

The system architecture of the Message boards is as Figure1 shows. There are two kind of storage system used in this task, Redis and Mongodb. The Redis layer served as in-memory cache to increase system throughput and latency performance. Also, the listen command for the message boards is implemented by the Redis notification function. The Mongodb layer served as the backing store of the data, i.e. all the messages from different boards in the application.

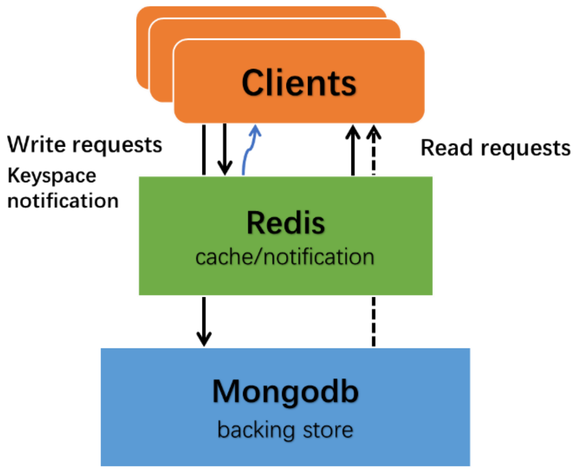


Figure1 System architecture

2.2 Design considerations

There are a few design considerations why I choose those two databases.

1, Redis is a key value database, it is a good fit to implement a message application like our message boards. We simply treated every message as a key-value pair, and the key can be generated using the board name plus request order or timestamp. Besides, Redis has notification support to easily implement listen interface in our case. So, in a functional perspective, using Redis is enough for implementing the message boards application.

However, the problem of using Redis alone is persistency since Redis is known for in-memory key value database. Although Redis provide persistency feature, it is prone to lose data for system/power failure.

2, To solve the persistency issue, I come up with using Mongodb, since Mongodb is a persistent database, i.e. every write will go into disk. For implementation simplicity consideration, we can use Mongodb alone to implement the message boards (may be use Redis notification function for listen command or simply using polling method). However, using Mongodb alone for all the read/write requests lead to another issue, i.e. performance. Since Mongodb is a persistent

database, when data volume grows to certain threshold (much bigger than memory can hold), the read performance will decrease significantly, which degrades the user experience of the application.
3, In order to address the performance issue, I come up with using Redis as a cache layer (this is similar as using Memcached as a cache layer in the lecture). The write requests will go to both Redis and Mongodb (potentially this can be done in parallel) similar as a write-through cache. The read requests will first consult the cache, i.e. Redis and see if there is data for read command. If some message got evicted from Redis, it will go to the Mongodb to retrieve the data. Due to the natural of locality, this architecture has much better read performance than using Mongodb as data store alone.
4, Some implementation details. I used Redis's atomic increament to generate suffix for each message. Board name plus the suffix composes as key in Redis. For Mongodb, each message is stored as a document with board name, id (counter suffix) and message itself.

I use the Redis notification mechanism to implement the listen command for the message boards application.

2.3 How application level operations translate

1) Select. The select command doesn't interact with data store, it simply change a global variable for read/write/listen commands.

2) Write. The write command will write to both Redis and Mongodb. (Since it may occur a read of that message immediately after that write, also, write immediately to Redis is used for notification feature for listen).

For all the boards, the application will store a counter record correspond to that board name. (key as full board name). At the beginning of execute write command, the application will first increase the counter (atomically) and get the current counter for further storing the data. For Redis, each write form a key-value pair, the key is formed as boardname_xxx, suffix is get from the atomic increment function in Redis and the data is write message itself. For Mongodb, the document is formed as {'board':boardname, 'id': counter, 'message':write_message}.

After write success, the application will print success information on screen.

3) Read. The read command first check board name performed by select command to do error handling. If the board is selected, it uses the board information to retrieve messages. There is a counter information stored in Redis, key is the full board name and data is the counter value (also as total number of messages). The application will first get the number of messages in this board and get data from Redis first using the board name plus suffix as keys. This is done using the Redis pipeline() basically asynchronized way to improve performance. After that, it checks all the messages returned from Redis. If certain message doesn't exist, it will go to Mongodb to retrieve data and re-set the message record to Redis for further use. Finally print out all the messages in that board to screen.
Noticed, if failure happens all the data in Redis lost including the important counter information. The application can scan Mongodb entirely and rebuild the counter.

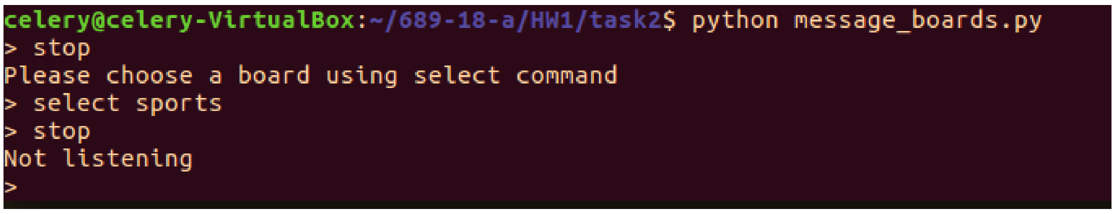
4) Listen. The listen function is implemented using the notification feature in Redis as stated above. Since every write command will write a key value pair to Redis, the listen user will got notification for that particular key and the application will get that record from Redis and print to the listening user.

3 Prototype

3.1 Code explanation

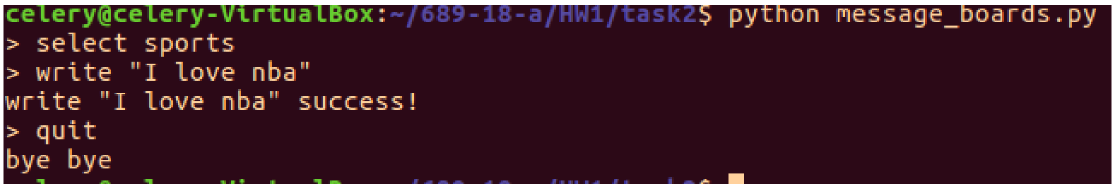
The code is simple enough to read directly since it's written in python. Also, I provide detailed comments in the code. Besides, section 2.3 provide detailed explanation of how the code works case by case. Thus, here I only provide some implementation details.

1) For stop listen command. I implement to method, you can stop listen by input stop command or using Ctrl+D (EOF).
2) For error handling for stop command. If there is no board select and you stop, the application will output please select board message. If there is board select but no listening and you stop, it will output not listening. Like below:



```
celery@celery-VirtualBox:~/689-18-a/HW1/task2$ python message_boards.py
> stop
Please choose a board using select command
> select sports
> stop
Not listening
>
```

3) For quit command, you can type quit any time, even if you are listening, you can still type quit and quit the application. Like below:



```
celery@celery-VirtualBox:~/689-18-a/HW1/task2$ python message_boards.py
> select sports
> write "I love nba"
write "I love nba" success!
> quit
bye bye
```

3.2 Instructions to run the code

1) Start Redis locally with notification enabled

Install Redis in your local machine and add Redis binary directory to your environment path, then in the shell run (or you can run in the binary directory similarly)

$ redis-server --notify-keyspace-events KEA

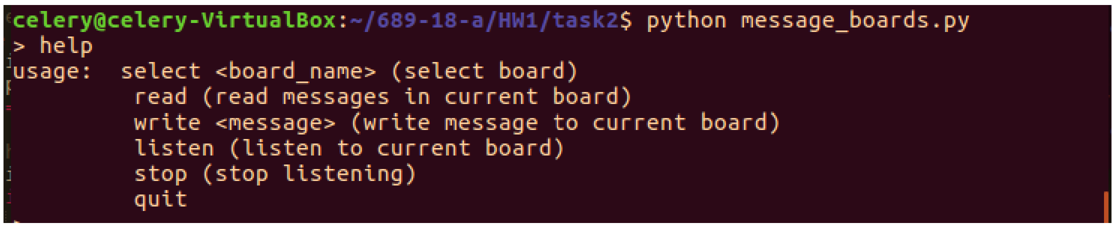
2) Run application

In the application directory, run

$ python message_boards.py

3) Usage instructions

The application provides enough error handling and are friendly to user.



```
celery@celery-VirtualBox:~/689-18-a/HW1/task2$ python message_boards.py
> help
usage:  select <board_name> (select board)
         read (read messages in current board)
         write <message> (write message to current board)
         listen (listen to current board)
         stop (stop listening)
         quit
```