Cloud Computing Homework 1: Databases

Task 2 Detailed Report

Task 2 Specifications:

The AggieFit team wanted to add message boards to encourage the employees to interact and exchange information with each other. They intended to categorize the message boards by topics. The employees could view and add messages to boards that they were interested in.

The team took recommendation from experts and were told that they could use two databases based on their requirements: MongoDB and Redis. The experts were not told about the application.

The AggieFit team wanted to know where the databases fit in their message board application.
The basic functionality they desired was as follows:
(i) Select Message Board: Choose a message board to read and write from. All operations are performed only after the message board is selected.
(ii) Read: Should be able to read all messages on the selected message board.
(iii) Write: Write the message into the selected message board.
(iv) Listen to Updates: Waits to get real time updates on the selected message board. If any other user writes on this message board, you should be able to read it immediately. Unlike read, the user does not have to explicitly pull the data. If the user is listening on a board, they should receive updates immediately after the board is updated.

I was asked to propose a System Architecture focused on the Databases for the Real-Time Chat Application and to build a prototype of the same.

Approximate Time Taken to Complete the Task: 2 Days
The time includes connection, configuration and integration of MongoDB and Redis using Python, formulation and proposal of a System Database Architecture and also the prototype to be built as per the client instructions.

Objective1) System Architecture Proposal

Building a chat application requires a communication channel over which a client can send messages that are redistributed to other participants in the chat room. This communication is popularly implemented using the publish-subscribe pattern (PubSub), where a message is sent to a centralized topic channel. Interested parties can subscribe to this channel to be notified of updates. This pattern decouples the publisher and subscribers, so that the set of subscribers can grow or shrink without the knowledge of the publisher.

PubSub is implemented on a backend server, to which clients communicate using WebSockets. WebSockets is a persistent TCP connection that provides a channel for data to be streamed bidirectionally between the client and server. With a single-server architecture, one PubSub application can manage the state of publishers and subscribers, and also the message redistribution to clients over WebSockets.

A single-server architecture is valuable to illustrate the communication flow. However, a multiserver architecture helps to increase reliability and create elasticity to horizontally scale your application as the number of clients grow.

In a multiserver architecture, a client makes a WebSocket connection to a load balancer that forwards traffic to a pool of servers. These servers are responsible for managing the WebSocket connections and data streamed over them. Once a WebSocket connection is established with a PubSub application server, that connection is persisted and data streams to the application in both directions. The load balancer distributes requests for a WebSocket connection to healthy servers, meaning that two clients can establish a WebSocket connection to different application servers.

Because multiple applications manage the client WebSocket connections, the applications must communicate among themselves to redistribute messages. This communication is necessary because a message might stream over a WebSocket up to one application server and need to be streamed down to a client connected to a different application server. You can satisfy this requirement for shared communication among applications by externalizing the PubSub solution from the applications that are managing client connections.

A persistent connection is established through the load balancer between each client and WebSocket server. A persistent connection is also established between the WebSocket server and the PubSub server for each subscription topic, which is shared among all clients.

Redis is an open source (BSD licensed), in-memory data structure store, used as a database, cache and message broker that features Pub-Sub support. It also supports data structures such as strings, hashes, lists, sets, sorted sets with range queries, bitmaps, hyperloglogs and geospatial indexes with radius queries.

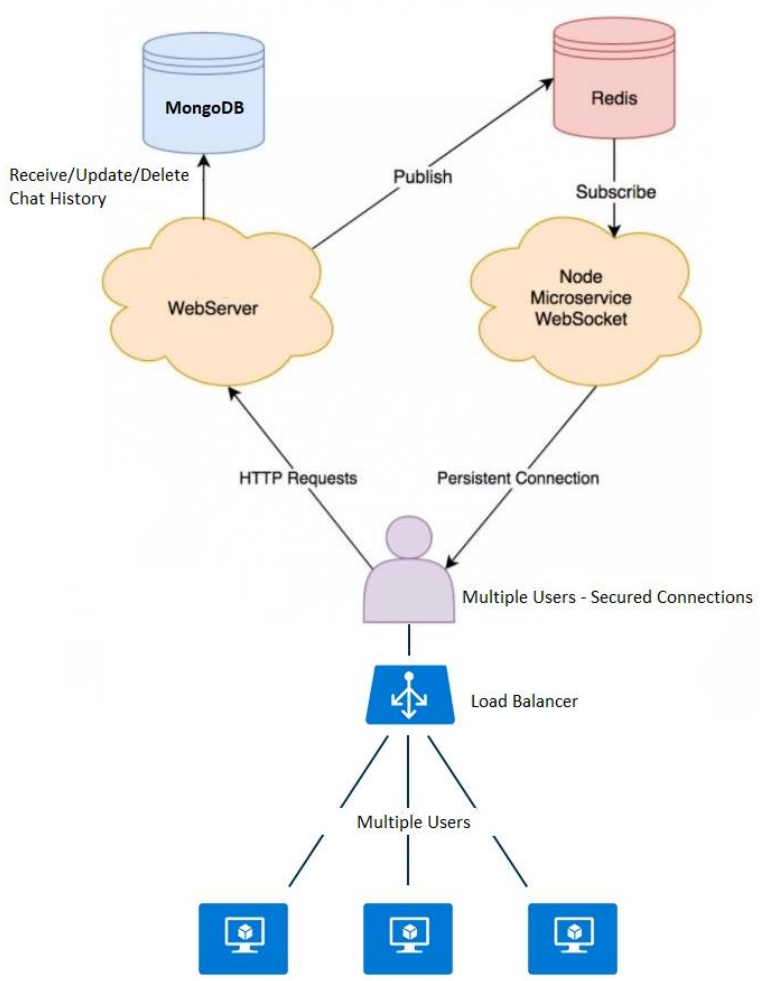
MongoDB, being a Collection and Key-Value based Document Data Model has quite a lot of advantages. MongoDB offers Dynamic Schemas which makes it very easy to add objects and fields at the same time, especially unstructured data, into the database. Also, MongoDB offers large storage capacity and high speed along with greater efficiency and reliability on handling large volumes of data.

So, we can use the pub-sub feature of Redis for the real-time chatting. However, it is not advisable to use Redis for maintaining chat history as it is not possible to store large volumes of data in an in-memory database which is not highly scalable too. Hence, the need arises to use another database model, i.e. MongoDB for maintaining the chat history as it can handle large volumes of unstructured data.

To listen to the Redis publications and create a WebSocket server, we can create a microservice using NodeJS. When initialized, it creates a WebSocket using the socket.io library, subscribing as a Redis listener.

The microservice works like a messaging router. After receiving messages from Redis, it checks if the recipient is connected via WebSockets and forwards the content.
Messages sent to unconnected recipients are discarded, but, when users log in to chat, they will receive messages through HTTP requests provided by any relevant web server. Similarly chat history can be viewed/updated/removed through HTTP requests using the same web server.

The Design Diagram of the System Architecture for the application is as follows:



Real-Time Chat Architecture

Objective2) Chat Application Prototype

We need to implement a simple prototype that demonstrates basic functionality of the system. For now, a simple command line interface has been created. Test scenarios involve multiple users accessing the boards in a particular order.

The Python Script for the prototype is '**message_boards.py**'.
Note: Before running the script, the user must be connected to the redis server.
The Script can be run on a Command Line Interface as: python message_boards.py

Script:

```
from mongo_connect import connectMongo
import constants
import pymongo
import json
import pprint
import redis
collection = connectMongo()
dir(redis)
r = redis.Redis()
subscribing = False;
topic = "";

while True:
	try:
		if subscribing:
			print("Subscribed")
			for item in p.listen():
				print(item)
		cmd = raw_input('Waiting For Command: ')
		print(cmd)
		cmd_parts = cmd.split(" ")
		cmd_parts[0] = cmd_parts[0].lower()
		print(cmd_parts)
		if cmd_parts[0] == "select":
			topic = cmd_parts[1].strip(' \t\n\r')
		elif cmd_parts[0] == "read" and topic != "":
			RQ = collection.find({ "_id": topic })
			for data in RQ:
				pprint.pprint(data)
			##res = r.lrange(topic, 0, -1) cd
			##print res
		elif cmd_parts[0] == "read" and topic == "":
			print("No Message Board Selected to Read")
```

```
        elif cmd_parts[0] == "write" and topic != "":
                to_set = (' '.join(cmd_parts[1:])).strip(' \t\n\r')
                ##r.rpush(topic, to_set)
                ##res = r.publish(topic, to_set)
                ##print res
                if to_set != "":
                        try:
                                collection.insert([{ '_id' : topic, '_msgs' : [to_set] }])
                                res = r.publish(topic, to_set)
                                print res
                        except:
                                collection.update(
                                { "_id": topic },
                                { '$push': { "_msgs": to_set } }
                                )
                                res = r.publish(topic, to_set)
                                print res
                else:
                        print("No Message - Please Try Again")
        elif cmd_parts[0] == "write" and topic == "":
                print("No Message Board Selected to Write")
        elif cmd_parts[0] == "listen" and topic != "":
                subscribing = True;
                p = r.pubsub()
                res = p.subscribe([topic])
                print res
        elif cmd_parts[0] == "listen" and topic == "":
                print("No Message Board Selected to Listen")
        elif cmd_parts[0] == "flush":
                r.flushdb()
                collection.remove({"_id": topic})
        elif cmd_parts[0] == "reset" and topic != "":
                topic = ""
        elif cmd_parts[0] == "reset" and topic == "":
                print("No Message Board Selected to Reset")
        elif cmd_parts[0] == "quit":
                break;
        else:
                print("Input Format Wrong");

except KeyboardInterrupt:
        subscribing = False
except:
        print("Oops! Something Went Wrong!!! - Kindly Check Instruction Manual Again")
```

The following instructions/operations are supported in the prototype:

(i) Select Message Board: select <board_name>

- Here a User Selects a Message Board/Chat Channel.

(ii) Read: read

- If No Board is Selected, then an Error Message is displayed.
- If a Board is Selected, then a document with all messages in the board gets displayed in an array.

(iii) Write: write <message>

- If No Board is Selected, then an Error Message is displayed.
- If a Board is Selected, then either a new document (i.e. new board_name) gets created with the message stored in an array or an old document board (i.e. existing board_name) gets updated with the message appended to the existing array.
- Also, on every successful insertion/updation of the document, a real-time message gets published in real-time to the subscribers, i.e. listeners of the board.

(iv) Listen to Updates: listen

- If No Board is Selected, then an Error Message is displayed.
- If a Board is Selected, then a User gets subscribed to the board and is ready to receive any real-time updates on the channel.

(v) Stop Listening to Updates: <Ctrl-C>

- A User can come out of Listening Mode by Keyboard Interrupt <Ctrl-C>

(vi) Reset Application: reset

- The board/channel gets reset, i.e. set to null

(vii) Clean Message Board/Clear RAM: flush

- If No Board is Selected, then an Error Message is displayed.
- If a Board is Selected, then on flush, the document of the selected board/channel with all its fields and records gets removed and also the RAM is cleared.

(viii) Quit Application: quit

- The User Exits from the Application.