Design Document: *MQTT Server & Client*

Dane Gottwald

1 Overall Goals

The goal is to create a server and corresponding client that communicate to each other via the MQTT publish-subscribe network protocol. The server shall be designed in C++ using boost libraries and the mqtt_cpp library while the client will be designed in a Python script that uses Mosquitto to communicate to the server.

1.1 Client

The client's specific goal is to generate a random count value between 1-9 inclusively and publish the result to the server at random intervals. Additionally, we want the client script to have multiple instances of itself running to simulate multiple various connections to the server.

1.2 Server

The server's specific goal is to be a data processor application that accepts incoming connections from the client, processes the count value that is published, and generates a 3 minute average of the counter data that is gathered in sum from each of the various connecting clients. The average that is taken is calculated from the total counter data gathered divided by the number of clients that published up to that point.

2 Client Design

The client is going to revolve around the use of Mosquitto's 'mosquitto_pub' command. In order to have the client simulate various connections while also sending the 'mosquitto_pub' command to the command line, the multiprocessing and subprocess libraries in Python can be used. Multiprocessing will handle the simulation of the multiple clients and subprocess will allow us to start a subprocess for the command to run.

```
import multiprocessing, subprocess

def loop:
      while True:
            sleep(#generate random interval to sleep)
            ran = #generate random [0-9]
            subprocess.run("mosquitto_pub -h IP -t TOPIC -m " + ran)

processes = []
for _ in range(#number of processes):
      processes.append(Process with target=loop)

# Start the processes

# Wait for each process to finish
```

This is the general structure of the Python script that will give us the desired effect of simulating multiple clients sending random count data at random intervals.

3 Server Design

Worthwhile documentation for Server design:

- https://github.com/redboltz/mqtt_cpp
    - Documentation: https://redboltz.github.io/contents/mqtt/
- https://www.boost.org/doc/

For MQTT communication in C++, we are going to use the mqtt_cpp library from above along with the provided examples in the repository.

NEXT PAGE

The steps needed to achieve the goal for the Server are as follows:

1. Create mqtt::server instance
    a. Requires: Endpoint: ip::tcp::endpoint
        i. Requires: Protocol: boost::asio::ip::tcp::v4()
        ii. Requires: Port: 1883
    b. Requires: IO Context: boost::asio::io_context
2. Set mqtt::server instance handlers
    a. set_error_handler
    b. set_accept_handler
3. Set relevant client endpoint handlers
    a. set_error_handler
    b. set_connect_handler
    c. set_publish_handler
    d. set_disconnect_handler
4. Setup asynchronous timer that will calculate the 3 minute average of counter data
    a. steady_timer
        i. Requires: IO Context
        ii. Requires: chrono time
    b. Create callback function for timer
    c. Call async_wait(), which takes a function pointer to callback function
5. Call listen() on mqtt::server object
6. Call run() on IO Context object

The Server will be ready to accept connections after we call [5] listen() and then the main thread will block on [6] run() to poll any server events that may occur, such as a client connecting. The timer using async_wait() will run asynchronously so it can continue processing the 3 minute average while the main thread handles client connections, requests, disconnects, etc.

My idea behind keeping everything thread safe and not creating any possible race conditions or data races when the timer and main thread are accessing the same data is to take advantage of atomic data types, which are provided by boost. Using an atomic struct to contain the cross-thread data I will need, I can create a lambda function for the timer that captures the struct by reference while also doing the same for the set_publish_handler. When a publish occurs, increment the client count by 1 and the count data by the value sent as a message. When the timer is actuated every 3 minutes, compute the average of the counter data (sum count data / client count) then reset both values back to 0 allowing for the cycle to continue.

