Implementation of Multi-Threading Content Flow with MQTT Broker

Yelin Mao, Hongkuan Yu
UNI: ym3000, hy2819

Fall 2024

Contents

1 Overview

MQTT has become a widely adopted application layer protocol in the Internet of Things (IoT) ecosystem due to its lightweight design and efficiency. An example of its application is in industrial control systems, where MQTT is used to control programmable logic controllers (PLCs). Beyond industrial operations, modern Internet usage and everyday electronic devices increasingly demand high-speed data exchange, content storage and distribution, and frequent processing of substantial volumes of data in real-time. Hence, implementing multi-threading techniques becomes critical to meet these requirements while remaining cost-effective and ensuring seamless content flow. This report examines the integration of multithreading data exchange techniques with MQTT brokers at the software level to enhance performance and meet the stringent demands of IoT applications.

2 Introduction to MQTT Protocol

Message Queuing Telemetry Transport, abbreviated as MQTT, is a publish-subscribe machine-to-machine messaging protocol that enables efficient and reliable data exchange even under constrained network conditions. Originating from IBM and now standardized by OASIS, MQTT has become a fundamental protocol of the IoT ecosystem due to its low overhead, simplicity, and scalability.

MQTT runs on a client-broker architecture, defining two main components: a message broker and clients. An MQTT broker acts as a central server that receives client messages and routes them to the appropriate recipients. Clients can be simple microcontrollers (e.g. PLCs) or complex servers that connect to the broker over a network using the MQTT library.

Data in MQTT are organized hierarchically into **topics**. When a publisher (one of the clients) has new information to share, it sends a control message containing the data to the broker. The broker then forwards this data to all other clients subscribed to the relevant topic. This decoupled design ensures that publishers do not need to know the number or location of subscribers, while subscribers do not need information about publishers.

If a broker receives a message for a topic with no active subscribers, it will be discarded unless the message is marked as "retained". Retaining messages allows the broker to store the latest data for a topic, ensuring that new subscribers receive the latest messages immediately after subscribing. In addition, publishers can specify a default message to be sent to subscribers if the publisher unexpectedly disconnects. While clients interact only with brokers, larger systems can integrate multiple brokers to exchange data based on topics for their respective subscribers.

The lightweight design of MQTT is reflected in its control messages, which can be as small as 2 bytes, but can hold up to 256 megabytes of data when necessary. The protocol defines 14 message types to manage tasks such as client connections, data publication, acknowledgments, and connection supervision, making MQTT highly adaptable to a wide range of IoT applications [1].

3 Multi-Threading v.s. Multi-Processing

Our software project adopts a "one process connects to one broker with multiple subthreads handling separate topics" architecture, designed to efficiently act as a bridge between the database and the MQTT broker. This design aligns well with the straightforward nature of MQTT's subscribe and publish functionalities, as the **Introduction to MQTT Protocol** section described. Both subscribers and publishers operate as subthread clients, following a standardized workflow: they connect to the MQTT broker and interact with specific topics.

In the publishing workflow, a client retrieves the required message from the database and publishes it to the broker by calling the publish function. Subscribers, on the other hand, adopt a passive role. Once connected to the broker, they monitor their subscribed topics for incoming messages. When a message is received, it is directly recorded into the database, requiring no additional action. This approach ensures seamless and efficient data exchange between the MQTT broker and the database.

Employing multiple threads within a single process is crucial for handling the software's high input and output demands. In contrast, a multi-process approach will limit each process to a single client connected to one broker and one topic. While this approach simplifies the code, it imposes scalability constraints. For example, with a 6-core CPU, running 1 process per core will allow only 6 processes, or 6 clients, to operate in parallel. However, by adopting a multi-threading approach, where each process utilizes 3 threads, the same 6 processes can support 18 clients concurrently. Figure 1 illustrates this concept clearly, as shown below.

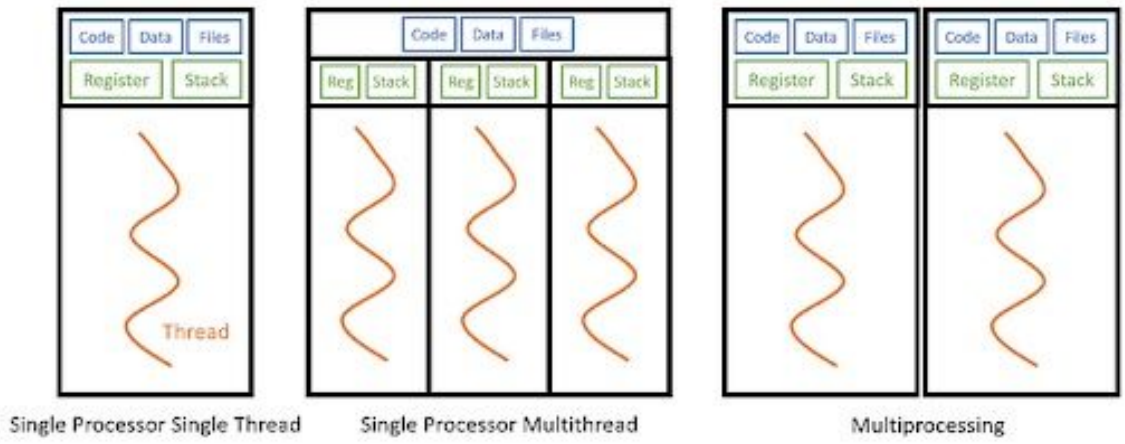

Figure 1: Multithreading v.s. Multiprocessing

Multi-threading concurrency functions like a pipeline rather than occurring simultaneously, whereas multi-processing parallelism happens simultaneously. In a single process with three threads running on one core, each thread spends a substantial amount of time on Input or Output (I/O) operations, such as the latency involved in receiving messages. These operations are relatively slow compared to modern CPUs' high processing speed and low latency. Therefore, while one thread is busy with I/O, the core can switch to another thread to perform tasks such as writing received messages to the database, initiating publishing, or verifying thread activity. Figure 2 below provides a clear visual representation of this concept.

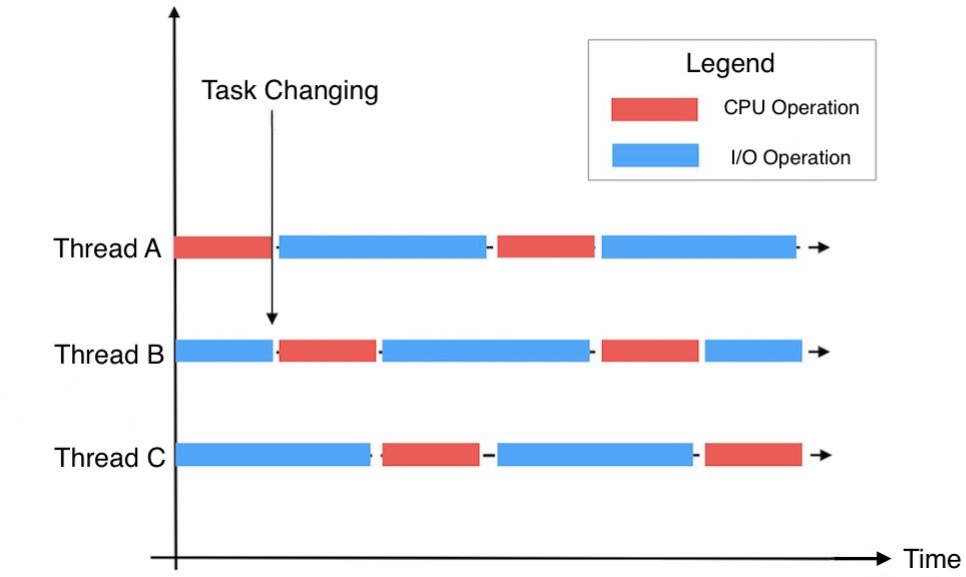

Figure 2: Concurrency

By wisely applying mutexes to manage thread synchronization, multi-threading proves to be a more resource-efficient and scalable solution than multi-processing in our project, making it highly suitable for the demands of IoT applications.

4 Software Environment Set Up

As this project is a software program, selecting an appropriate programming language is crucial. Initially, we considered using Python due to its straightforward syntax and coding logic. However, Python's Global Interpreter Lock (GIL) prevents true multi-threading, limiting its performance in concurrent environments. Additionally, while Python automatically handles memory management, it consumes more resources than languages like C, C++, or Java. After thorough evaluation, we chose C++ as our programming language because it is Turing-complete, offering greater flexibility than C and providing a more comprehensive feature set compared to Java.

Developing a software project on Linux provides a highly efficient and comfortable environment, so we selected Ubuntu's latest long-term support version, 24.04, as our operating system [2].

We selected the Qt6 framework for C++ 14 because it offers a variety of well-established libraries that greatly simplify our development process [3].

We utilized CLion by JetBrains as our compiler [4]. Similar to PyCharm for Python, CLion features a comparable interface and provides valuable tools that enhance coding and debugging efficiency for C++.

We chose Eclipse Mosquitto as our MQTT broker due to its lightweight, open-source nature [5]. It offers sufficient functionality for our project and is easy to set up and build on Ubuntu.

5 Multi-Threading Techniques

5.1 QMqttClient and QThread Class

The two most frequently utilized classes in our project are **QMqttClient** [6] and **QThread** [7].

5.1.1 QMqttClient Class

The `QMqttClient` class offers essential functions, such as subscribe and publish, which can be accessed when an `QObject` type is instantiated through inheritance from `QMqttClient`. In real-world applications, certain objects, like PLCs, are real devices that do not require instantiation. These devices act as subscribers to receive commands from the MQTT Broker and as publishers to send useful information back to the MQTT Broker. In our implementation, we created virtual publisher subthreads to send commands and virtual subscriber subthreads to receive and process the data between the database and the MQTT Broker.

5.1.2 QThread Class

The `QThread` class allows each thread to have its event loop, as shown in Figure 3 below.

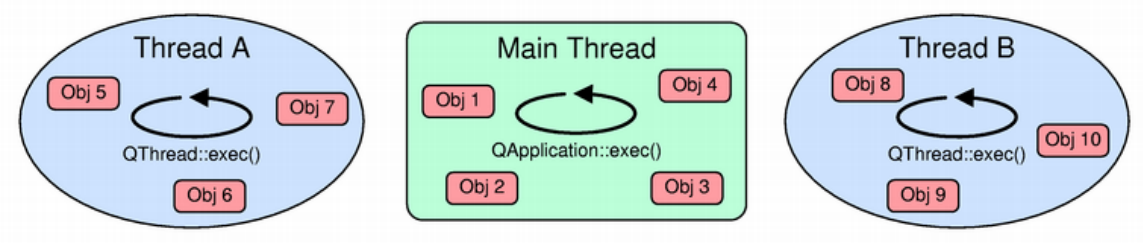

Figure 3: How QThread Work

The main thread starts its event loop using `QCoreApplication::exec()`. For subthreads, the event loop can be initiated using `QThread::exec()`. Similarly to `QCoreApplication`, the `QThread` class also provides a `exit(int)` function and a `quit()` slot to manage the event loop.

An event loop within a thread enables the use of specific non-GUI Qt classes that rely on the presence of an event loop, such as `QMqttClient` class. Additionally,

it allows signals from any thread to be connected to slots within a specific thread, facilitating inter-thread communication.

An instance `QObject` is considered to reside in the thread where it was created. The event loop of the corresponding thread handles events that target that object. The thread, in which a `QObject` resides, can be determined using the `QObject::thread()` function.

5.2 Code Structure

The code we wrote contains:

```
main.cpp
MqttThreadConnector.cpp
MqttThreadPool.cpp
MqttReadThread.cpp
MqttWriteThread.cpp
```

with their header files (`.h`) respectively.

5.2.1 main.cpp

The code in `main.cpp` is shown below.

```
#include <QtCore>
//#include <iostream>
//#include <unistd.h>

#include "MqttThreadConnector.h"
#include "MqttThreadPool.h"
#include "MqttReadThread.h"
#include "MqttWriteThread.h"


using namespace std;

int main(int argc, char *argv[]) {

     QCoreApplication app(argc, argv);

     MqttThreadPool mtp(2, 2, "127.0.0.1", 1883);
```

```
    mtp.simpleTest();

    return QCoreApplication::exec();
}
```

The code in **main.cpp** is kept minimal, as writing all logic within this file is neither efficient nor aligned with object-oriented programming principles. Instead, we organized the project by placing the core logic into separate `.cpp` files, ensuring that each file is responsible for a specific set of related functions. The role of **main.cpp** is simply to instantiate a test object defined in other files. In this case, it sets up a thread pool by instantiating an object from **MqttThreadPool.cpp**, managed by a main thread, consisting of two publisher subthreads and two subscriber subthreads. The IP address `127.0.0.1` and Port `1883` correspond to the localhost MQTT broker, allowing us to test the code on our laptops conveniently.

5.2.2 MqttThreadConnector.cpp

The code in `MqttThreadConnector.cpp` is long. Therefore, its corresponding header file, `MqttThreadConnector.h`, is presented below to help clarify our design and ideas.

```
#pragma once
#include <QThread>
#include <QMutex>
#include <QtMqtt/QMqttClient>
#include <QTimer>

class MqttThreadConnector : public QThread {
    Q_OBJECT
public:
    explicit MqttThreadConnector(QObject* parent,
                                 const QString &brokerIp,
                                 int brokerPort);
    ~MqttThreadConnector() override;
    [[nodiscard]] QString getThreadId() const;

    // a function that lets the outside know whether this is the
    ↪  specific thread they want to look for
    [[nodiscard]] bool specificThread(const QString& strThreadId)
    ↪  const;
```

```
    // mainThread makes the sub-thread to run. Runtime can
    ↪  modify here. Unit: millisecond.
    void reqStart(int _workTimeMS=500);

    // mainThread makes the sub-thread to stop. Stop time can
    ↪  modify here. Unit: millisecond.
    void reqStop(int stopTimeMS=5000);

    // a function that lets the outside check the working status
    bool isWorking(int maxBusyTimeMS=10000);

    // To print out some useful information
    [[nodiscard]] QString getInfo() const;

    enum connectionState{
        closed=0,
        connecting,
        firstConnect,
        threadExit
    };
    Q_ENUM(connectionState)

protected:
    //subclass must override this virtual method.
    virtual void onMessageReceived(const QByteArray& message,
    ↪  const QMqttTopicName& url); //Read

    virtual void getSubscribeInfo(QString& url, quint8& qos);
    virtual void getPublishInfo(QString& url, quint8& qos,
    ↪  QByteArray& msg);

    virtual void onDataSendStart(qint32 id, const QString& url,
    ↪  const QByteArray& msg);
    virtual void onDataSentFinish(qint32 id);

    // Read/Write: client.subscribe(url, qos);
    ↪  client.publish(url, message, qos, false);
```

```
    // Write: quint32 id = client.publish(url, message, qos,
    ↪  false);

private:
    connectionState state;

    void run() override; // sub-thread entry code, called by
    ↪  Operating System
    void installEventHandle(QMqttClient& client);
    void install_errorChanged(QMqttClient& client);
    void install_stateChanged(QMqttClient& client);
    void install_messageReceived(QMqttClient& client);
    void install_messageStatusChanged(QMqttClient& client);

    void routineWork();     //called by timer
    void doSubscribe();
    void doPublish();

    void setLivingTime();               //Critical data; need
    ↪  mutex
    QDateTime getLivingTime();          //critical data; need
    ↪  mutex
    bool threadNeedExit();
    void setThreadExit();

    QString brokerIp;
    int brokerPort;

    //Qt::HANDLE threadId;
    QString threadId;

    QDateTime livingTime;               //Critical data; need
    ↪  mutex
    QMutex mutex;
    QMqttClient* threadClient= nullptr;          //ref the client
    ↪  created in stack of sub-thread by the run method
    QTimer* threadTimer= nullptr;
```

```
    int workTimeMS = 500;                          //default 500
    ↪  milliseconds; can be overwritten by _workTimeMS above
};
```

The code in **MqttThreadConnector.cpp** implements the core logic and functions for managing subthreads. This class inherits from `QThread`, which in turn inherits from `QObject`. Since the subscribe and publish operations share several common functions, we packaged these shared functions into this class for better reusability (this is why we chose to name this class "Connector").

Different subscribers and publishers must run on separate subthreads, which are created and managed by the main thread instantiated in **main.cpp**. Since subthread management involves shared functions, we implemented these common operations in this class and made them public, allowing other classes to inherit and use them. We also implemented several public functions that allow external to retrieve useful information about the status of each thread.

We left the protected functions unimplemented in this class because they are business logic rather than shared core functionalities. These functions were inherited and overwritten in **MqttReadThread.cpp** and **MqttWriteThread.cpp** to fulfill specific business logic. Here, the protected functions serve as interfaces only.

The private functions consist of shared operations for both subscriber and publisher subthreads; however, they are not intended to be accessed externally. Keeping them private allows for better memory management. These functions handle tasks such as managing connections to the broker, error detection, and watchdog operations for each subthread. It is important to note that the functions `void install_messageReceived` and `void install_messageStatusChanged` are not shared with publisher subthreads and are specifically used by subscriber subthreads. When a subscriber receives a message from a subscribed topic, it emits a signal to trigger the slot connected to these two functions. Publishers, on the other hand, cannot trigger them. Since these functions act as a finite state machine for processing received messages, we considered them part of the core logic and implemented them directly within this class.

5.2.3 MqttThreadPool.cpp

The code in `MqttThreadPool.cpp` is long. Therefore, its corresponding header file, `MqttThreadPool.h`, is presented below to help clarify our design and ideas.

```
#pragma once
```

```
class MqttThreadPool {

public:
    explicit MqttThreadPool(int numReadThread,
                            int numWriteThread,
                            const QString &brokerIp,
                            int brokerPort);

    ~MqttThreadPool();

    void simpleTest();

private:
    QList<MqttReadThread *> readThreadList;
    QList<MqttWriteThread *> writeThreadList;

    int numReadThread;
    int numWriteThread;
    QString brokerIp;
    int brokerPort;

    Qt::HANDLE mainThreadId;

    void createThread();

    void killDeadThread();
    void killAllThread();
};
```

This class does not inherit from any other class because it is solely designed for subthread management. Following a similar approach as mentioned above, the public function `void simpleTest` acts as an interface for external components to call and utilize, as demonstrated in **main.cpp**. The private functions, on the other hand, handle the internal logic of the thread pool, which includes generating lists to manage them. Since these private functions do not need to be accessed externally, they are packaged to ensure efficient memory management.

5.2.4 MqttReadThread.cpp

The code in `MqttReadThread.cpp` is shown below.

```
#include <QTimer>
#include "MqttReadThread.h"

MqttReadThread::MqttReadThread(QObject *parent,
                               const QString &brokerIp, int
                               ↪ brokerPort) :
                               ↪ MqttThreadConnector(parent,
                               ↪ brokerIp, brokerPort) {
}

MqttReadThread::~MqttReadThread() = default;

void MqttReadThread::onMessageReceived(const QByteArray &message,
↪  const QMqttTopicName &url) {
    qDebug() << "==========> MqttReadThread(" << getThreadId() <<
    ↪  "):::messageReceived ==========> msg = " << message
             << ", topic = " << url
             << "\n";
}

void MqttReadThread::getSubscribeInfo(QString& url, quint8& qos)
↪  {
    url = "test/topic";
    qos = 0;
}
```

This class implements the business logic for the subscriber subthread. It inherits from the **MqttThreadConnector.cpp**, allowing us to pass the `brokerIp` and `brokerPort` parameters to the functions defined in **MqttThreadConnector.cpp** and establish a connection to the broker.

As described in the **MqttThreadConnector.cpp** above, two functions related to the subscription process are overwritten in this class. The `void onMessageReceived` function prints an indication upon receiving a message; the `void getSubscribeInfo` function specifies the topic to subscribe to.

Some may observe that we fixed the IP and port variables in the constructor while keeping the topic and QoS (Quality of Service) variables flexible in `void`

`getSubscribeInfo` (though they are also fixed for testing purposes in the current code). The reason is that we decided each process should connect to a single MQTT broker, as mentioned in **Multi-Threading v.s. Multi-Processing** section. In case multiple brokers are involved (requiring multiple processes) and some experience connection issues, it becomes easier to pinpoint the disconnected processes and restart them entirely. In contrast, detecting and fixing connection problems at the subthread level is far more difficult and inefficient. Additionally, a single broker can manage multiple topics for clients to subscribe and publish, each with its respective QoS. This influenced our decision to make the topic and QoS variables flexible.

These two protected functions currently implement relatively simple business logic for testing purposes; potential improvements will be discussed in **Conclusions and Future Improvements** section.

5.2.5 MqttWriteThread.cpp

The code in `MqttWriteThread.cpp` is shown below.

```
#include <QTimer>
#include <sstream>

#include "MqttWriteThread.h"

MqttWriteThread::MqttWriteThread(QObject *parent,
                                 const QString &brokerIp,
                                 int brokerPort) :
                                 ↪ MqttThreadConnector(parent,
                                 ↪ brokerIp, brokerPort) {
}

MqttWriteThread::~MqttWriteThread() = default;

void MqttWriteThread::getPublishInfo(QString& url, quint8& qos,
↪ QByteArray& msg) {
    url = "test/topic";
    qos = 0;
    msg = QByteArray("Hello World!");
}
```

```
void MqttWriteThread::onDataSendStart(qint32 id, const QString&
↪  url, const QByteArray& msg)  {
}

void MqttWriteThread::onDataSentFinish(qint32 id) {
}
```

This class implements the business logic for the publisher subthread. It inherits from the **MqttThreadConnector.cpp**, allowing us to pass the `brokerIp` and `brokerPort` parameters to the functions defined in **MqttThreadConnector.cpp** and establish a connection to the broker.

As outlined in the **MqttThreadConnector.cpp** above, three functions related to the publication process are overwritten in this class. The `void getPublishInfo` function is responsible for initiating the publish operation, while the other two functions are intended for database-related operations. Currently, these three protected functions either implement simple business logic or are left blank for testing purposes; potential enhancements will be addressed in the **Conclusions and Future Improvements** section. Notice that the IP and port variables are fixed in the constructor, while the topic, QoS, and message variables remain flexible for the same reasons discussed in the **MqttReadThread.cpp** section.

6 Results

Part of the code from `MqttThreadPool.cpp` is shown below.

```
void MqttThreadPool::createThread() {
    int createReadThread = numReadThread -
    ↪  static_cast<int>(readThreadList.size());
    int createWriteThread = numWriteThread -
    ↪  static_cast<int>(writeThreadList.size());

    if (createReadThread <= 0 and createWriteThread <= 0) {
        return;
    }

    //Create Read Thread
    for (int i=0; i < createReadThread; ++i) {
        qDebug() << "Read ThreadPool, controlled by (" <<
        ↪  mainThreadId << "), ..... creates read thread: " <<
        ↪  i;

        auto rt = new MqttReadThread(nullptr, brokerIp,
        ↪  brokerPort);

        readThreadList.append(rt);

        if (i == 0) {
            rt->reqStart(2000);
        }
        else
            rt->reqStart(500);
    }

    //Create Write Thread
    for (int i=0; i < createWriteThread; ++i) {
        qDebug() << "Write ThreadPool, controlled by (" <<
        ↪  mainThreadId << "), ..... creates write thread: " <<
        ↪  i;
```

```
        auto wt = new MqttWriteThread(nullptr, brokerIp,
        ↪  brokerPort);

        writeThreadList.append(wt);

        if (i == 0) {
            wt->reqStart(2000);
        }
        else
            wt->reqStart(2000);
    }
}

void MqttThreadPool::killDeadThread() {
    // record which sub-thread is dead.

    QList<int> killIndex;
    QList<MqttReadThread*> toKillRead;
    QList<MqttWriteThread*> toKillWrite;

    for (auto rThread: readThreadList) {
        if (rThread->isWorking(600)) {
            //qDebug() << "********** threadId= " <<
            ↪  pThread->getThreadId();
            continue;
        }
        qDebug() << "Read ThreadPool, controlled by (" <<
        ↪  mainThreadId << "), is ready to kill: " <<
        ↪  rThread->getInfo();
        toKillRead.append(rThread);
    }

    for (auto wThread: writeThreadList) {
        if (wThread->isWorking(600)) {
            //qDebug() << "********** threadId= " <<
            ↪  pThread->getThreadId();
            continue;
        }
```

```
            qDebug() << "Write ThreadPool, controlled by (" <<
            ↪ mainThreadId << "), is ready to kill: " <<
            ↪ wThread->getInfo();
            toKillWrite.append(wThread);
        }

        //remove toKillRead threads from readThreadList
        for(auto killRead : toKillRead) {
            readThreadList.removeOne(killRead);
        }

        //remove toKillWrite threads from writeThreadList
        for(auto killWrite : toKillWrite) {
            writeThreadList.removeOne(killWrite);
        }

        // stop threads in the toKillRead list
        for (auto rTh: toKillRead) {  //iterate over the list
            rTh->reqStop();
            delete rTh;             //delete the memory pointed by
            ↪ the variable pThread
        }
        toKillRead.clear();

        // stop threads in the toKillWrite list
        for (auto wTh: toKillWrite) {  //iterate over the list
            wTh->reqStop();
            delete wTh;             //delete the memory pointed by
            ↪ the variable pThread
        }
        toKillWrite.clear();

    //    qDebug() << "toKillRead=" << toKillRead;
    //    qDebug() << "readThreadList=" << readThreadList;
    }
```

For simple testing, we set the `maxBusyTimeMS` to 600 milliseconds and passed it to the `bool isWorking` function for all subscriber and publisher subthreads. This function checks the status of each subthread by comparing the current system times-

tamp with the last recorded living timestamp, which is updated using the `void setLivingTime` function. If the timestamp difference exceeds 600 milliseconds, the subthread is considered "dead." In such cases, the thread-pool stops the nonresponsive subthread and creates a new one to replace it.

We designed two test cases to validate the functionality of our code thoroughly. As discussed in the **main.cpp** section, creating just two publisher and two subscriber subthreads is sufficient for testing purposes. Also, we kept the Eclipse Mosquitto MQTT broker running in the background as a local host service.

6.0.1 Test Case 1: 2000-500-2000-2000

The first test case is shown in the code above. In this test, the first subscriber subthread reports its "alive" status every 2000 milliseconds, while the second subscriber subthread does so every 500 milliseconds. Similarly, the first publisher subthread reports its "alive" status every 2000 milliseconds, and the second publisher subthread every 2000 milliseconds. We referred to this as the "2000-500-2000-2000" case.

Subscriber subthreads have a slightly more dynamic mechanism compared to publisher subthreads. While both types of subthreads report their "alive" status at predefined intervals, subscriber subthreads update their "alive" timestamp whenever they successfully receive messages. For subscriber subthreads, message reception can occur more frequently than the reporting interval, requiring them to update their status in real-time during operation. When message flow is light, subscriber subthreads fall back to reporting their "alive" status periodically to ensure they remain responsive. In contrast, publisher subthreads strictly follow the preset intervals to scan the database and publish messages. If there is little or no content to publish, publisher subthreads remain mostly idle, relying on periodic status updates to indicate that they are still active.

Hence, the expected outcome for this case is as follows: only one 500 ms subscriber subthread will continue running, while other subthreads will be terminated and recreated since its 2000 ms period exceeds the 600 ms threshold. The terminal output is shown in Figure 4.

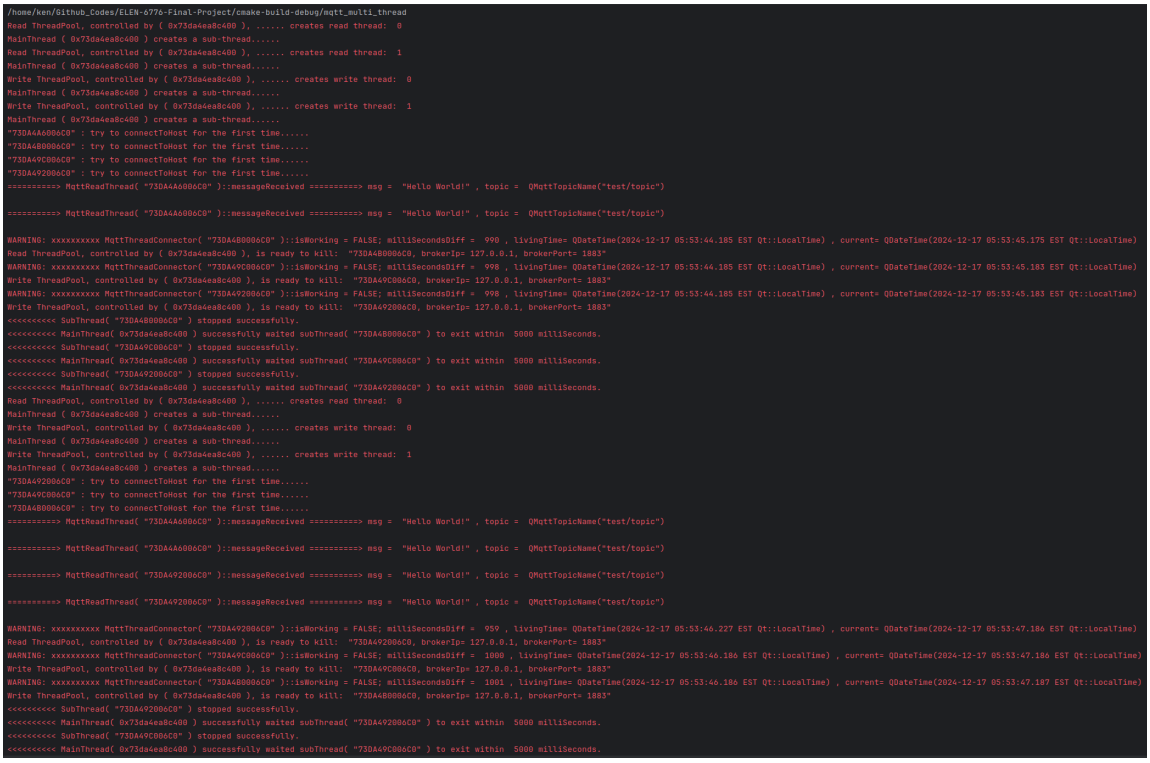

Figure 4: Test Case 1: 2000-500-2000-2000 Output

The output met our expectations. As shown in Figure 4, the main thread had a thread ID of 73DA4EA8C400 and successfully created four subthreads with IDs 73DA4A6006C0, 73DA4B0006C0, 73DA49C006C0, and 73DA492006C0. The fact that all five thread IDs were distinct confirms that the multithreading techniques were correctly implemented in our software. Once the subthreads were created, they attempted to connect to the MQTT broker. Upon establishing the connection, they began performing their respective tasks.

The subscriber subthread with ID 73DA4A6006C0 successfully received messages from the two publisher subthreads, providing two corresponding indications. It is worth noting that the other subscriber subthread was also expected to display the received messages with two indications; however, during the first trial, this did not occur. Instead, the thread pool detected the other subscriber subthread as "dead," terminated it, and then regenerated it during the second trial. This behavior also aligned with our expectations. As discussed in the **Multi-Threading v.s. Multi-Processing** section, multithreading concurrency functions similarly to a pipeline.

When the program initially started, the core performed several operations, which may cause delays. Consequently, when switched to the other subscriber subthread to execute its subscription tasks, the 600-millisecond "alive" threshold may already be exceeded, leading to its termination and regeneration.

In the second trial, the thread pool successfully terminated and regenerated the subthreads. Notably, the new subthreads reused and reassigned their previous thread IDs: 73DA4B0006C0, 73DA49C006C0, and 73DA492006C0 were all reused. Interestingly, the thread with ID 73DA492006C0, which was a publisher subthread in the first trial, was reassigned as a subscriber subthread after its termination and regeneration.

The second trial also met our expectations. The output showed four message received indications, with each of the two subscriber subthreads contributing two indications. As the code above specifies, the "Number Zero" subscriber subthread has a live period of 2000 milliseconds, while the publisher subthreads operate every 2000 milliseconds, which exceeds the 600-millisecond threshold. Consequently, the subscriber subthread with ID 73DA492006C0 does not receive any messages afterward and is unable to refresh its alive timestamp, leading to its termination and regeneration in the third trial.

We conducted multiple trials over one minute. Although the terminal output was too lengthy to capture in a screenshot, all trials followed the same pattern as observed in the second trial. This consistent behavior confirmed that our program operated correctly and successfully applied the multithreading techniques.

6.0.2 Test Case 2: 2000-500-2000-500

In the second test case, the first subscriber subthread reports its "alive" status every 2000 milliseconds, while the second subscriber subthread does so every 500 milliseconds. Similarly, the first publisher subthread reports its "alive" status every 2000 milliseconds, and the second publisher subthread every 500 milliseconds. We referred to this as the "2000-500-2000-500" case.

The expected outcome for this case is as follows: both subscriber subthreads will continue running, while one 2000 ms publisher subthreads will be terminated and recreated since its 2000 ms period exceeds the 600 ms threshold. The terminal output is shown in Figure 5.

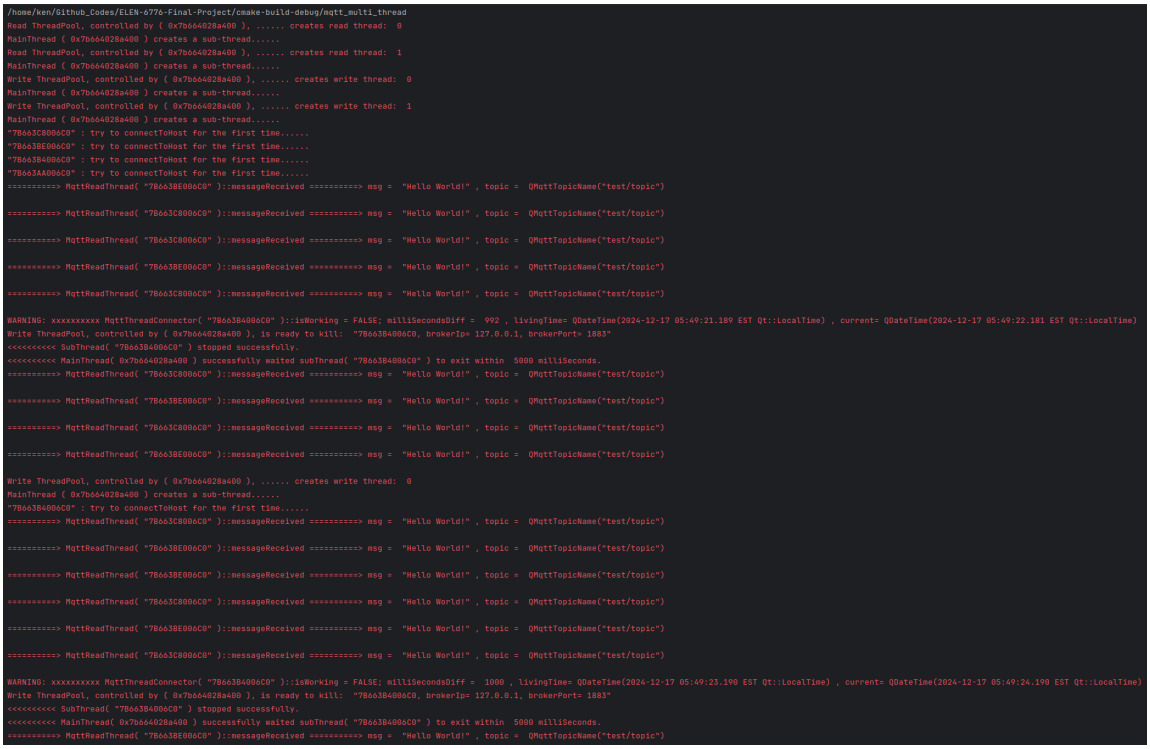

Figure 5: Test Case 2: 2000-500-2000-500 Output

The output met our expectations. As shown in Figure 5, the main thread had a thread ID of 7B664028A400 and successfully created four subthreads with IDs 7B663C8006C0, 7B663BE006C0, 7B663B4006C0, and 7B663AA006C0. The fact that all five thread IDs were distinct confirms that the multithreading techniques were correctly implemented in our software. Once the subthreads were created, they attempted to connect to the MQTT broker. Upon establishing the connection, they began performing their respective tasks.

The output showed five message received indications from the two subscriber subthreads. As expected, one of the subscriber subthreads posted at least two message indications. Although we initially anticipated four message indications, the presence of one additional indication from a subscriber subthread was entirely acceptable, as the same reason in the **Multi-Threading v.s. Multi-Processing** and **Test Case 1: 2000-500-2000-2000** sections.

In the second trial, the thread pool successfully terminated and regenerated the 2000 ms publisher subthread, reusing and reassigning its previous thread ID,

7B663B4006C0. This trial also aligned with our expectations. While the first subscriber subthread operated on a 2000 ms interval, the 500 ms publisher subthread published a message every 500 milliseconds. Since all subthreads were connected to the local host, latency remained minimal. Consequently, the first subscriber subthread could receive messages from the 500 ms publisher subthread and update its "alive" timestamp during message reception within the 600 ms threshold.

We conducted multiple trials over one minute. Although the terminal output was too lengthy to capture in a screenshot, all trials followed the same pattern as observed in the second trial. This consistent behavior confirmed that our program operated correctly and successfully applied the multithreading techniques.

6.0.3 Test Case 3: MQTT Broker Crushed

In the third test case, we evaluated the robustness of our program. To handle a scenario where the process crashes, we considered implementing Ubuntu's crontab as a higher-level watchdog to monitor and restart the process. However, due to time constraints, we opted to skip this part. On the MQTT broker side, implementing error handling was more straightforward, as we utilized the available functions from the `QMqttClient` class. We incorporated various event and error handling mechanisms into the private functions of the **MqttThreadConnector.cpp** class, as discussed earlier.

Keeping all the code the same as in **Test Case 1: 2000-500-2000-2000**, we manually shut down the MQTT broker while the program was running to simulate a scenario where the broker crashed. The terminal output is shown in Figure 6.

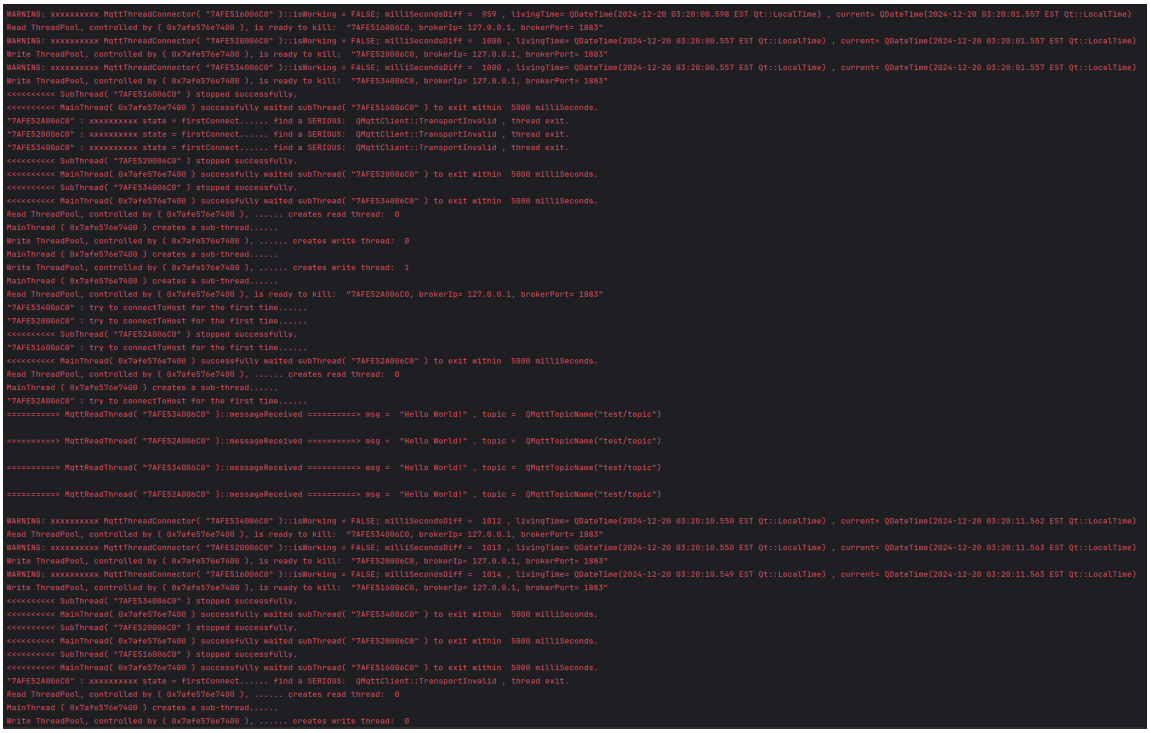

Figure 6: Test Case 3: MQTT Broker Crushed Output

The output demonstrated that the process did not crash despite the subthreads encountering connection errors. Instead, all subthreads were stopped by the thread pool orderly, which then regenerated new subthreads of the same type to attempt reconnection to the same MQTT broker. It is important to note that the watchdog for each subthread only starts after a successful connection to the MQTT broker. Once we manually restarted the MQTT broker, the newly generated subthreads successfully reconnected and resumed normal operation, functioning as expected, similar to the behavior observed in **Test Case 1: 2000-500-2000-2000**. This result confirms the strong robustness of our program.

7 Conclusions and Future Improvements

The **Results** section demonstrated that our core logic for implementing multithreading techniques and ensuring program robustness is solid. As our project and this report primarily focused on the core logic, we consider the program a significant success.

However, if more time were available, there are several areas where the program could be improved. First, we did not have sufficient time to implement an SQL database to fully achieve our intended goals. Since the program is designed to act as a bridge between a database and the MQTT broker, the absence of a real database makes our simulation results less comprehensive.

Second, due to the lack of a database, we had to hardcode the topics, QoS, and message variables within the overridden functions in **MqttReadThread.cpp** and **MqttWriteThread.cpp**. Additionally, some overridden functions in these classes were left blank. These functions were originally planned to interact with a database via REST API. In our design, the subscriber subthreads would retrieve different topic URLs and QoS levels from the database to perform specific subscriptions (`void getSubscribeInfo`). After receiving a message, instead of simply printing it, the subscriber subthreads would send the message back to the appropriate location in the database under the respective topics (`void onMessageReceived`). For the publisher subthreads, they were intended to fetch different topic URLs, QoS levels, and messages from the database (`void getPublishInfo`). With varying QoS levels, the publisher subthreads would notify both the subthread watchdog and the database that publishing was in progress (`void onDataSendStart`). Upon completing the publishing operation, they would inform the subthread watchdog about their status and update the database with a successful publishing indication (`void onDataSentFinish`). If a publisher subthread failed without reporting success, the watchdog would terminate and regenerate the subthread, allowing it to retry the same command and information fetched from the database. This mechanism would effectively prevent duplicate publishing commands.

In a nutshell, our project successfully demonstrated the implementation of multithreading techniques with strong program robustness, serving as a solid foundation for MQTT IoT applications. While the current version focuses on core logic, integrating a real SQL database and completing the planned database interactions would greatly enhance the functionality and practicality of the program. By enabling dynamic topic management, message handling, and fault tolerance through REST API integration, the program can efficiently bridge MQTT brokers and databases in real-world IoT scenarios. These improvements would make the program more scalable,

reliable, and capable of supporting the high demands of IoT systems, providing a clear path for future development to unlock its full potential.

References

[1] Wikipedia contributors, "Mqtt — Wikipedia, the free encyclopedia," 2024, [Online; accessed 12-December-2024]. [Online]. Available: https://en.wikipedia.org/w/index.php?title=MQTT&oldid=1257675205

[2] Ubuntu. Ubuntu: Enterprise Open Source and Linux. [Online]. Available: https://ubuntu.com/

[3] Qt. Qt | Tools for Each Stage of Software Development Lifecycle. [Online]. Available: https://www.qt.io/

[4] JetBrains. CLion: A Cross-Platform IDE for C and C++ by JetBrains. [Online]. Available: https://www.jetbrains.com/clion/

[5] E. Mosquitto. Eclipse Mosquitto^{TM} An open source MQTT broker. [Online]. Available: https://mosquitto.org/

[6] Qt6. QMqttClient Class. [Online]. Available: https://doc.qt.io/qt-6/qmqttclient.html

[7] ——. QThread Class. [Online]. Available: https://doc.qt.io/qt-6/qthread.html