International Journal for Multidisciplinary Research (IJFMR)

E-ISSN: 2582-2160 ● Website: www.ijfmr.com ● Email: editor@ijfmr.com

Priority Based Message Queue Processing Using Software Thread

Binoy Kurikaparambil Revi

Independent Research
binoyrevi@live.com

Abstract:
In the world of restful services for the web application, the backend or most commonly called core software is the brain of most real world applications. In this type of complex software system, the core software has a lot of high priority tasks which can be real time or near real time and it may not have much bandwidth to deal with sharing the data or messages to web services like restful services. Priority based Message Queue Processing Using Software Thread is a software implementation technique that can be immensely helpful to handle the data and messages from the core software that need to be transmitted without blocking the core software program execution.

Introduction:

Priority based Message Queue Processing Using Software Thread provides an extremely lightweight implementation using the software thread to the overall software architecture that can handle the data and messages from core software to transmit to various endpoints. This technique also provides a good strategy to handle priority and standard messages in separate queues. Handling priority and standard messages separately is a quite common use case in many complex real time applications. This is because there are critical messages or data that need to be transmitted immediately without sending those messages to the message queue and waiting for its turn to transmit.

Adding Messages to the Message Queues

Message Queues can be managed in different ways from simple First In First Out(FIFO) to complex Multiple Queues Processing(MQP). In the majority of the applications and in almost all real time applications, the Multiple Queues Processing is very much essential as the messages have a priority tag attached to it. Let's consider a real time system that issues standard data messages with a ‘Normal’ priority tag, Important notification messages with ‘High’ priority tag and Error Messages with ‘Critical’ priority tag. So we technically need to have 3 queues to process, let's name them ‘Normal’, ‘High’ and ‘Critical’ queues. Using a software thread we can process these Message queues based on a set of priority requirements for the application.

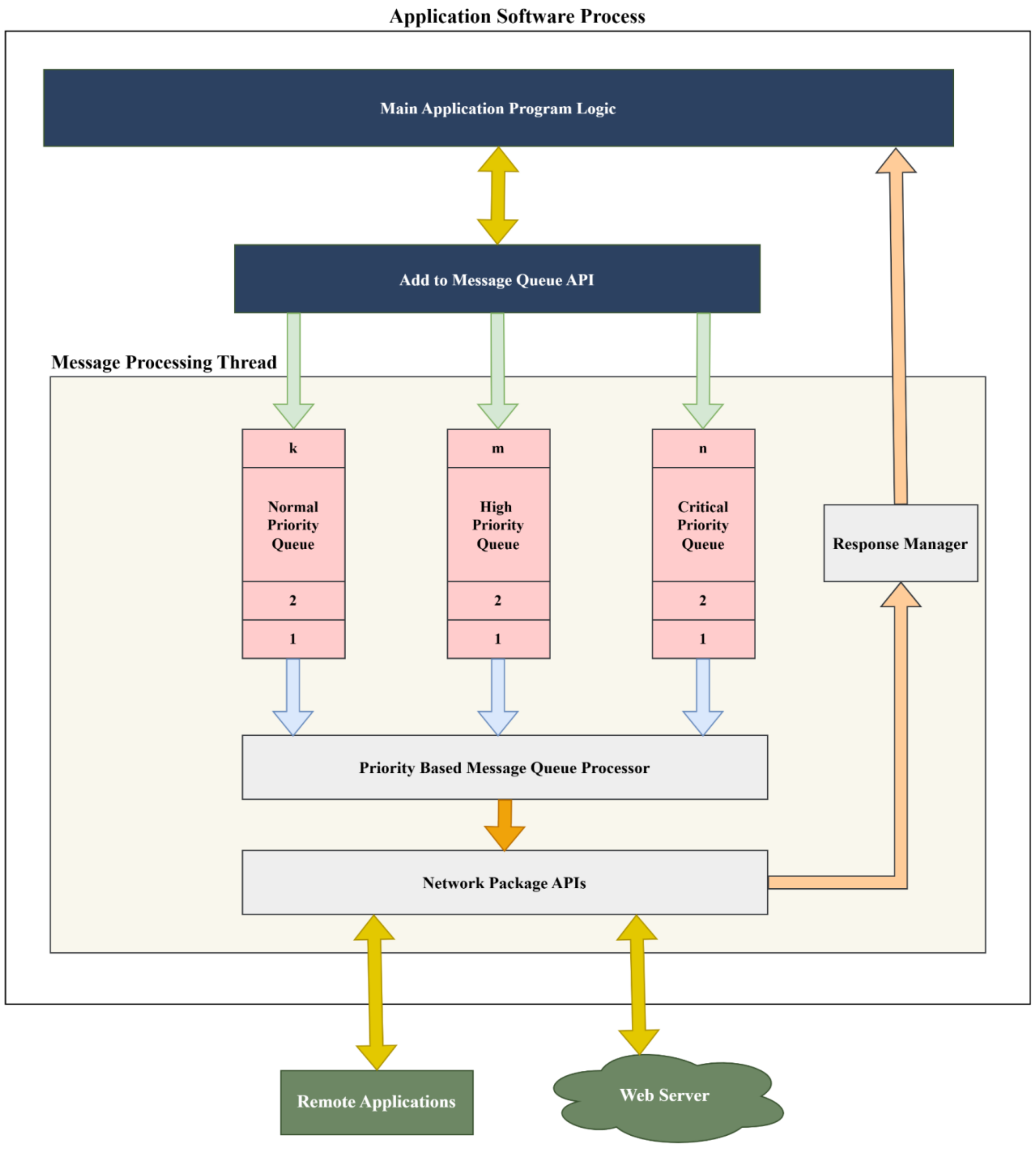

Figure 1: Priority Based Message Queue Processing Using Software Thread

Figure 1 gives an overall design on how the Priority based Message Queue Processing works in an application program. The main idea behind this design is that a software thread is a light weight process still attached to the main process and shares the same memory space with the process. As threads are lightweight, it uses less resources.

When a new message needs to be issued to a remote entity like a web server or a remote application, the main program just calls the API to add the message to the queue. For this the main program needs to pass the message as well as the priority information to the API as parameters. The API then checks the priority and adds the message to the appropriate queue. Another important task that API performs is to limit the

length of the queue to a predefined limit. If the limit is reached, typically the oldest message is removed and a newer message is added to the top of the queue. However this may be different depending on the application requirements. As an example, in some cases, the API no longer accepts the messages that need to be added to the critical message queue if the critical message queue is full.
As the API is not responsible for transmission and its job is just to add the messages to an in-memory queue, it is extremely fast in completing the execution and returning the control to the main application program. This makes this technique very handy to use in the real time applications.

Priority Based Message Queue Processor in action
Message queue processor is the key software function in the thread. This queue processor always keeps checking the queue for messages. If it finds messages in the queue, the queue processor immediately starts processing the messages. Once a message is processed, which means the message got transmitted to a remote application or a web server, the Message Queue Processor removes the message from the queue.
The most important and may be critical function of the Message Queue Processor is the order in which the queues are processed. One simple logic to do this is to complete transmission of all messages in the critical queues and then the high priority queues and finally the Normal priority queues. There may be scenarios like new message arrival on a high priority queue when a low priority queue is under processing. In this case generally the current processing is completed and then the Message Queue Processor switches to the high priority queue to process the high priority messages. As the thread execution is independent from the main program execution, a near parallel execution performance is achieved where the messages are sent to the endpoints while the main program executes its key functionality.

Network Package APIs and Response Manager
The network libraries are the underlying mechanism that can be used to perform the tasks of transmitting the package to the network. I recommend using QT Libraries to perform these tasks. The QNetworkAccessManager class provided by QT libraries allows us to create an object that allows applications, in this case the Message Queue Processor, to send the Restful API request to the network. This will help to transmit the messages from the queue to the network as POST requests.

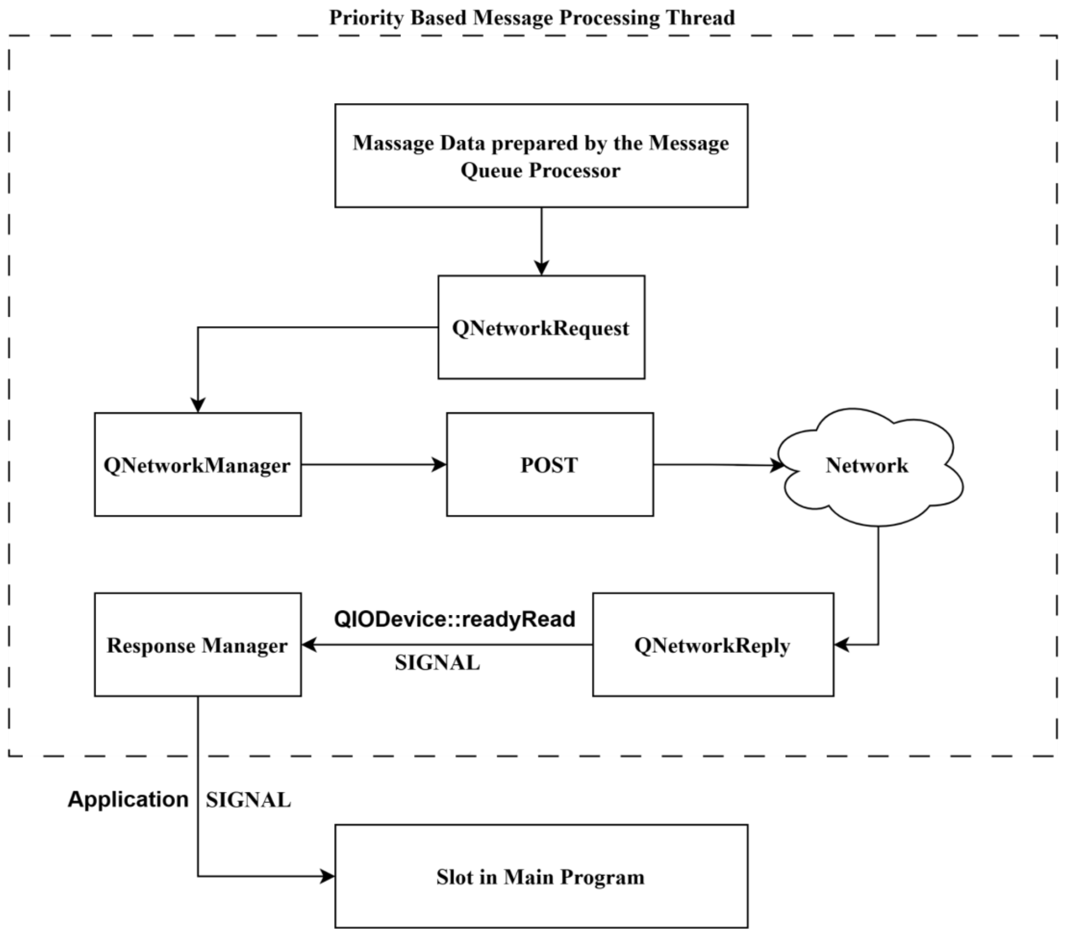

Figure 2: RestAPI Implementation using QT Network Libraries to transmit messages

A typical C++ implementation of the Restful service in a priority based network processing thread is given in Figure 2. A major benefit of using QT libraries to implement this thread is the fact that QT framework uses the signal slot mechanism to manage the data flow across different functions in the program. As in the above diagram, the QNetworkManger uses the QNetworkRequest to configure and pack the message and then post it to the network. The QNetworkReply object is configured to receive QIODevice::readyRead signal when there is a response available to read. This signal will invoke Response Manager and it is responsible to signal the main application process as per the business logic. Error handling signals are also available to catch the errors from QNetworkReply.

Conclusion:

Using the QT libraries and multi-threading mechanism, Priority based Message Queue Processing provides an efficient implementation technique to manage message transmission across the network in a thread that runs alongside the main core application. This technique reduces network communication overhead on the main application thus allowing it to focus on the core business logic. Overall, it adds Network messaging capability to the core backend program without a significant impact on its performance. One thing I would like to add to conclude this is that it will be extremely helpful to send health status signals from the thread to the main application program so that if the thread fails for any reason, the application can restart the thread with appropriate priority.

References:

1. K. B. Wheeler, R. C. Murphy and D. Thain, "Qthreads: An API for programming with millions of lightweight threads," 2008 IEEE International Symposium on Parallel and Distributed Processing, Miami, FL, USA, 2008, pp. 1-8, doi: 10.1109/IPDPS.2008.4536359.
2. B. Falsafi and D. A. Wood, "Parallel Dispatch Queue: a queue-based programming abstraction to parallelize fine-grain communication protocols," Proceedings Fifth International Symposium on High-Performance Computer Architecture, Orlando, FL, USA, 1999, pp. 182-192, doi: 10.1109/HPCA.1999.744362.
3. Rischpater, R., Zucker, D. (2010). Beginning Qt Development. In: Beginning Nokia Apps Development. Apress. https://doi.org/10.1007/978-1-4302-3178-3_4