MQTT++ v5

Evelyn G. Coronel

Programación Orientada a Objetos
Instituto Balseiro

(15 de diciembre de 2020)

I. MQTT SIMPLIFICADO EN C++

En este trabajo se basó en la implementación simplificada del protocolo de MQTT utilizando los temas de la materia. Las siguientes palabras son parte de la nomenclatura de la implementación del protocolo:

- Tópico (topic): Es un tema en el cual se publican mensajes, por ejemplo: en el tópico `casa/cocina/luces/` pueden publicarse mensajes de apagado/encendido de las luces suscritas al tópico.
- Cliente: El cliente se puede suscribir a varios tópicos y publicar mensajes en los mismos.
- Mensaje: El mismo se publica en un tópico y el broker se encarga de mandar el mensaje a los clientes suscritos al tópico en cuestión.
- Broker: se encarga de mandar los mensajes entre los clientes.

A. Librería `mqtt.hpp`

Esta librería están almacenadas las variables reservadas para el protocolo, dentro del `namespace mqtt`. También se implementó una clase que asigna valores de identificación (ID) únicos a los clientes.

B. Librería `mqtt_errors.hpp`

Esta librería está en el `namespace mqtt_errors` y contiene los posibles errores que puede aparecer en el algoritmo.

1. Clase `error`

Esta clase es hija de `std::runtime\_error` y con las misma se implementaron el resto de las clases de error, listadas a continuación:

- Clase `MQTT_ERR_CONN`: Cliente no disponible.
- Clase `MQTT_ERR_DEQUE`: Cola de mensajes llena.
- Clase `MQTT_ERR_DEQUE_EMPTY` : Cola de mensajes vacía.
- Clase `MQTT\_ERR\_MESS`: Nadie está suscrito al topic.
- Clase `MQTT\_ERR\_QOS` : Prioridad desconocida.

C. Librería `mqtt_message.hpp`

Implementado dentro del `namespace mqtt_message` están implementadas la función `topic_match_sub` la clase `mqtt_message::message`.

1. Función `topic_match_sub`

Está función recibe un string y una lista de strings, y compara si el string es igual o está contenido dentro de alguno de los strings de la lista, devuelve `true` si encuentra una coincidencia o `false` en caso contrario.

La función acepta los argumentos de la siguiente forma: `topic_match_sub(<string,forward_list<string>>)`

2. Clase `mqtt_message::message`

Cada mensaje tiene un tópico y un *payload*, donde el tópico es el tema del mensaje y el payload es el mensaje en sí: por ejemplo un mensaje puede publicarse en el tópico `/casa/luces/` con el payload de `"ON"` para encenderse.

También tiene un tag de prioridad, que dice decide cuando pronto deben ser publicados: la prioridad alta indica que deben ser publicados inmediatamente mientras que la prioridad normal anexa el mensaje al final de la cola de espera.

El mensaje también tiene la opción de retención, si el mensajes debe ser retenido con `retain=true`, si el mensajes no es recibido por nadie se mantiene en la cola hasta que existe algún cliente que los escuche.

Los métodos de esta clase son los siguientes:

```
- message(<string mensaje="",
           string topic="/",
           bool retain=false,
           short priority=mqtt::NORMAL)
```

Esta es la función que instancia una variable `message`. Al inicializar el mensajes se puede inicializar o no los atributos del mismo.

- `set_payload(<string>)` y `get_topic()`

  Se puede definir y obtener el payload del mensaje.

- `set_topic(<string>)` y `get_payload()`

  Se puede definir y obtener el tópico del mensaje.

- `is_retain()`

  Esta función devuelve `true` si el mensaje debe ser retenido, `false` caso contrario.

- `set_Priority(<short>)` y `get_Priority()`

  Se puede definir y obtener la prioridad del mensaje.

D. Librería `mqtt_client.hpp`

En esta librería están implementadas las clases utilizada para instanciar objetos que se comporten con un cliente. La misma está dentro del `namespace mqtt_client`.

1. Clase `mqtt_client::client_virtual`

En esta clase está implementada de tal forma que la misma y sus hijos no puedan copiarse. También se declaran funciones virtuales puras para que luego van a ser implementadas en las clases hijas.

2. Clase `mqtt_client::client`

Esta clase es heredada de la clase `client_virtual`. Se asigna un ID al azar, usando `client::get_id()` y `client::set_id(<unsigned int>)` cuando se conecta al broker que transmite los mensajes

Un cliente puede estar conectado a un solo server ya que al conectarse/desconectarse la variable `mqtt_client::Connected` cambia `mqtt::CONNECTED` / `mqtt::DISCONNECTED`. El estado del cliente se obtiene mediante la función `client::isConnected()`.

Un cliente puede estar suscrito a varios topics al mismo tiempo, la función `client::subscribe(<string>)` agrega un nuevo topic a una lista miembro de la clase `client`, la función `client::get_topic()` devuelve dicha lista y con la función `client::unsubscribe(<string>)` podemos desuscribir al cliente de algún topic

El cliente solo puede estar conectado a un servidor al mismo tiempo. Tiene la opción de hacer que sus mensajes siempre vayan por defecto, al inicio de la cola de mensajes con `mqtt_client::set_Priority(mqtt::HIGH)`.

La función `mqtt_client::client::reply()` debe ser implementada por cada tipo de cliente, ya que cada uno va a tener una forma de responder distinta.

E. Librería `mqtt_server.hpp`

La librería implementa una cola de mensajes con control de acceso, la misma está dentro del `namespace mqtt_server`

1. Estructura `mqtt_server::publisher`

Esta es una estructura que contiene de mensaje y la dirección del cliente que publicó el mensaje.

2. Clase `mqtt_server::server`

La clase tiene estos tres atributos para el control de acceso a la doble cola `deque` del tipo `publisher std::deque< publisher > message_deque`:

- `broadcasting_condition`
- `server_thread`
- `server_mutex`

Los mismos son `protected` para poder acceder a ellos mediante las clases heredadas como variables privadas. El `copy construct` y el `move construct` no son accesibles para no mover ni copiar el servidor.

Podemos inicializar el servidor con una variable `unsigned int` de la siguiente manera `server(unsigned int i)`, para definir la cantidad máxima de mensajes en la cola, por defecto la máxima cantidad de mensajes es 10.

- `set_timeout(<chrono::milliseconds>)`

  Esta función modifica el tiempo en el que esta prendido el servidor, este valor tiene un mínimo de 1 milisegundo.

- `set_interval(<chrono::milliseconds>`

  Esta función modifica el tiempo en el se transmiten dos mensajes consecutivos, el valor predeterminado es de 100 milisegundos entre mensajes.

- publisher `pop_message`

  Esta función obtiene el primer mensaje en espera de la cola.

- publisher `get_message`

  Esta función obtiene el primer mensaje en espera de la cola, también verifica que la cola no este vacía, si lo está devuelve `nullptr`.

- `append_message_from(<void *, mensaje *>)`

  Esta función agrega el mensaje de un cliente a la cola de mensajes, teniendo en cuenta la prioridad del mensaje. En la implementación se utiliza un lock

en el mutex del servidor para controlar el acceso a la cola de mensajes, de esta forma evitar conflicto con la función `pop_message`.

Para evitar un conflicto con un *include* recursivo, se utiliza un puntero del tipo `void *` para referenciar al cliente.

- `append_message(<mensaje *>)`

  Es un mensaje para todos los clientes desde el servidor, entonces el cliente es `nullptr`

- `broadcast_message()` Esta función es virtual pura en esta clase, se implementa en la clase hija `broker`. La función que publica los mensajes a todos los clientes.

- `start_broadcasting()` Esta función tiene en cuenta si el broker va a mandar mensajes durante un rango de tiempo definido o constantemente hasta terminar el programa. Por defecto, el envío de mensajes se hace por tiempo indefinido.

  - `constant_broadcasting()`: Verifica que se haya modificado el valor predeterminado nulo de la variable `server_timeout`, si no fue modificado, el servidor hacer un broadcasting constante.
  - `timeout_broadcasting()`

    Caso contrario, se realiza el broadcasting durante el tiempo determinado

- `running_thread()`

  Inicializa el thread del servidor para correr en paralelo con el thread padre. Esta función se ejecuta cuando se ejecuta la función `start_broadcasting`.

- `stop_broadcasting`

  Para la transmisión de mensajes y termina el thread del servidor.

F. Librería `mqtt_broker.hpp`

La librería está con el `namespace mqtt_broker` y se basa en el la librería `mqtt_server.hpp`.

1. Clase `mqtt_broker`

Esta clase es una clase hija de `mqtt\_server::server`, como atributos la clase tiene una lista con las direcciones de los clientes conectados al broker, un generador de IDs, además de un `mutex` y un `condition_variable` para manejar la lista de clientes, además del `mutex` y `condition_variable` para la cola de mensajes heredada de la clase padre.

Los métodos de esta clase incluyen a:

- `sin_subs()`: Devuelve `False` si no hay clientes conectados, `True` caso contrario.

- `publish_from(<cliente *, mensaje*>)`

  Publica un mensaje proveniente de cliente a los demás clientes mediante el servidor. El mismo usa un lock en el mutex del servidor para modificar a cola de mensajes, y evitar un conflicto con la función `broadcast_message()`.

  - `publish(<mensaje*>)`

    Publica un mensaje del servidor a todos los clientes

- `connect(<cliente *, string>)`

  Conecta un cliente al servidor.

- `disconnect(<cliente *>)`

  Desconecta un cliente del servidor. El mismo usa un lock en el mutex de la lista de clientes para verificar que la cola de mensajes está vacía para desconectar a un cliente.

- `broadcast_message()`

  La función que publica los mensajes a todos los clientes. El mismo usa un lock en el mutex del servidor para modificar a cola de mensajes, y evitar un conflicto con la función `publish_from` y `publish`.

G. Ejemplos

1. *ejemplo_wikipedia.cpp*

En este ejemplo se implementa la imagen de wikipedia explicando el protocolo MQTT.

2. *ejemplo_QoS.cpp*

Se implementa un servidor con mensajes de distintas prioridades.