Implementación y simulación de un Broker MQTT

MQTT (Message Queueing Telemetry Transport) es un protocolo de conexión de máquina a máquina utilizado en IoT (Internet of Things). Es un protocolo liviano de envío y recepción de mensajes del tipo *publish-subscribe*.

Modelo publish-subscribe

En este modelo de comunicación existen productores de mensajes, llamados *publishers*, que generan mensajes que no son enviados directamente a receptores específicos, llamados *subscribers*, de los que puede haber varios o ninguno. Los *subscribers* expresan su interés en recibir mensajes de ciertos tópicos y sólo reciben estos mensajes, sin tener noción de los *publishers*, si es que existen.

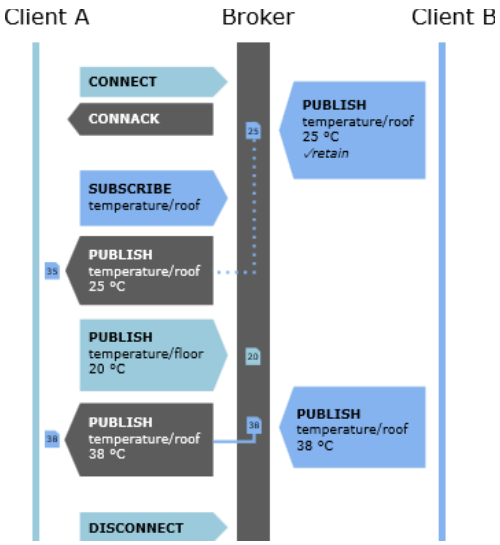
En estos sistemas, los mensajes son enviados por los *publishers* a un *broker* de mensajes intermediario. Los *susbscribers* registran sus subscripciones con el mismo *broker*. Ambos, *publisher* y *subscriber*, son clientes del *broker*. Cuando el broker recibe un mensaje con datos nuevos de un publisher redistribuye la información entre los clientes subscriptos a ese tópico. De esta manera los *publishers* no necesitan tener conocimiento de los *subscribers* ni viceversa.

Los mensajes recibidos por el *broker*, además de ser retransmitidos a los *subscribers* interesados, pueden ser retenidos si el *publisher* así lo indica. De esta manera el mensaje está inmediatamente disponible ante una nueva subscripción.

Mensajes MQTT a implementar

Existen distintos tipos de mensajes en MQTT, éstos tienen formatos especificados en el estándar. La idea es implementar clases de objetos que representen un subconjunto simplificado de ellos:

- CONNECT: primer mensaje enviado por un cliente a un *broker*. Lleva un nombre de usuario y su clave.
- CONNACK: acknowledge de la conexión enviado del *broker* a los clientes
- PUBLISH: indica el tópico del mensaje, su valor y un flag que indica retención del mensaje. En nuestra simplificación el tópico y el valor serán cadenas de caracteres. Es común que el tópico del mensaje tenga la forma “aaaa/bbbb”.
- SUBSCRIBE: indica que el cliente está interesado en recibir mensajes de un determinado tópico. En la realidad el tópico puede tener ***** como wildcard para indicar un filtro de múltiples tópicos que cumplan ese patrón.
- UNSUBSCRIBE: indica que el cliente no está más interesado en recibir mensajes de un determinado tópico.
- DISCONNECT: anuncio de desconexión de un cliente.



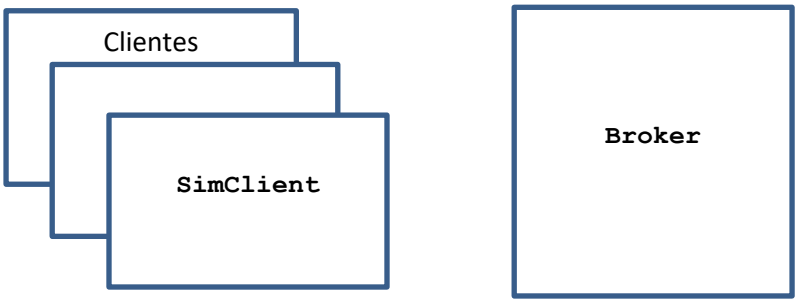
Esqueletos de definición de las clases de mensajes:

```
using TopicName  = string;
using TopicValue = string;

class Message {
    public:
        enum class Type { CONNECT, PUBLISH, SUBSCRIBE, UNSUBSCRIBE, DISCONNECT };
        Type getType() const;
        virtual Message *clone() = 0; // para evitar object splicing
    private:
        Type type;
     // ...
};

class ConnectMsg : public Message {
    private:
        string username;
        string password;
    // ...
};

class ConnAckMsg : public Message {
    public:
        enum class Status { CONNECTION_OK, LOGIN_ERROR, /* OTHER_ERRORS... */ };
    private:
        Status status;
    // ...
};

class PublishMsg : public Message {
    private:
        TopicName topic;
        TopicValue value;
        bool retain;
     // ...
};

class SubscribeMsg   : public Message { /* ... */ };
class UnsubscribeMsg : public Message { /* ... */ };
class DisconnectMsg  : public Message { /* ... */ };
```

Simulación de clientes y del broker

Se pide realizar la implementación de un *broker* que pueda recibir de y enviar mensajes a distintos clientes y simular su comportamiento. También será necesario implementar clientes para simular la dinámica de envío y recepción de mensajes. Para permitir la existencia de múltiples clientes simulados simultáneamente va ser necesario utilizar programación concurrente y por lo tanto sincronización para acceso a datos.

La idea es implementar un objeto singletón **`Broker`** y múltiples objetos **`SimClient`**.



Conexión de clientes al Broker

El **`Broker`** debería implementar un método **`BrokerOpsIF *registerClient(ClientOpsIF *)`** que utilizarían los clientes para establecer la conexión.

El **`Broker`** recibe una interface **`ClientOpsIF`** del cliente que utilizará para enviarle mensajes al cliente (PublishMsg y ConAckMsg).

```
// Interface implementada por el cliente para recibir los tópicos subscriptos
class ClientOpsIF {
    public:
        virtual void recvMsg(const Message &) = 0;
};
```

El cliente recibe del **`Broker`** una interface **`BrokerOpsIF`** que utiliza para enviar mensajes al **`Broker`**.

```
// Interface implementada por el Broker para enviarle mensajes
class BrokerOpsIF {
    public:
        virtual void sendMsg(const Message &) = 0;
};
```

Simulación de Clientes

Implementar una clase **`SimClient`** para los clientes simulados. Las tareas que realiza cada cliente deberían ejecutarse en un thread propio. Cuando se crea que un objeto **`SimClient`** se recibe una referencia al **`Broker`**, que le permitirá ejecutar el método **`registerClient`**.

```
class SimClient : public ClientOpsIF {
        thread simth;
        Broker &broker;
        virtual void runSim() = 0;
    public:
        SimClient(Broker &);
        void start();
    // ...
};

void SimClient::start()    // En pseudo C++
{
    simth = move(thread{&SimClient::runSim, this});
}

// Ejemplo de un cliente Publisher
class SimPublisher : public SimClient {
        void runSim();
    public:
        void recvMsg(const Message &);
    // ...
};

void SimPublisher::runSim()    // En pseudo C++
{
    BrokerOpsIF *brops = broker. registerClient(this);

    brops->sendMsg(ConnectMsg{"user", "pass"});        // CONNECT
    // esperar por CONNACK sin errores                 // wait CONNACK

    for( cierta cantidad de pasos ) {
         this_thread::sleep_for( un ratito random );

         PublishMsg m;
         fill m;

         brops->sendMsg(m);                            // PUBLISH
    }
    brops->sendMsg(DisconnectMsg{});                   // DISCONNECT
}
```

Broker

El *broker* del sistema debe recibir y enviar mensajes a distintos clientes.

Para poder realizar sus tareas, debería tener en su representación las siguientes entidades:

- Mantener un contenedor con los clientes conectados
- Conjunto de tópicos que están subscriptos, cada uno de ellos posiblemente subscripto por múltiples clientes. Tiene que ser eficiente buscar una subscripción por el nombre del tópico.
- Conjunto de tópicos retenidos. Tiene que ser eficiente buscar un tópico por su nombre.

Subscriptions y ReteinedTopics

Aparecen estas clases para representar los conceptos de subscripciones hechas por los clientes y los tópicos publicados con el requerimiento de ser retenidos para su inmediata utilización. Estos objetos están asociados al cliente que los generó. Una posible representación sería:

```
class Client;

struct Subscription {
    TopicName topic;
    Client *owner;
};

struct RetainedTopic {
    TopicName topic;
    TopicValue value;
    Client *owner;
};
```

Client

El **Broker** debe tener un objeto que represente un cliente conectado y que debería implementar la interface **BrokerOpsIF**. Se debe permitir que el procesamiento de los mensajes enviados por los **SimClient**s sea concurrente con su propia ejecución.

Una manera de permitir la ejecución independiente es en **BrokerOpsIF::sendMsg** encolar los mensajes y poner un thread a consumir los mensajes de la cola y procesarlos. Debe existir una cola por cliente para poder distinguir, por ejemplo, de donde vienen los pedidos de subscripción.

Los **Client**s al procesar los mensajes que le llegan, deben generar eventualmente las **Subscription**s y los **RetainedTopic**s, mantener el tracking de ellos y registrarlos con el **Broker**.

Un esqueleto para completar sería el siguiente. En particular le faltan todos los mecanismos de sincronización.

```
class Client : BrokerOpsIF {
    private:
        thread th;
        ClientOpsIF *cif;
        container<Subsciption *> subscriptions;
        container<RetainedTopic *> topics;
        cola<Message *> recvQueue;
    public:
        Client(ClientOpsIF *);
        void sendMsg(const Message &m) {
             // falta toda la sincronización...
             recvQueue.put(m.clone());
        }
    // ...
};
```

Una manera de permitir la ejecución independiente es en **BrokerOpsIF::sendMsg** encolar los mensajes y

poner un thread a consumir los mensajes de la cola y procesarlos. Debe existir una cola por cliente para poder distinguir, por ejemplo, de donde vienen los pedidos de subscripción.

Broker

El **`Broker`** debe implementar el método **`registerClient`**, que deberá instanciar un nuevo **`Client`** para atender al correspondiente **`SimClient`**. También deberá tener los caches para acceso búsqueda rápida de subscripciones y tópicos retenidos.

```
class Broker {
        container<Client *> clients;
        // caches para búsquedas eficientes
        container<Subsciption *> subs_cache;      // Múltiples subscripciones para un tópico
        container<RetainedTopic *> topics_cache;
    public:
        BrokerOpsIF * registerClient(ClientOpsIF *);
    // ...
};
```

Lista de sitios con información sobre MQTT:

- https://en.wikipedia.org/wiki/MQTT
- http://mqtt.org/
- https://en.wikipedia.org/wiki/Publish%E2%80%93subscribe_pattern
- http://docs.oasis-open.org/mqtt/mqtt/v3.1.1/os/mqtt-v3.1.1-os.pdf