ECE 448/528
Application Software Design

Lecture 13. Eclipse Paho MQTT Client
Spring 2025

Won-Jae Yi, Ph.D.

Department of Electrical and Computer Engineering
Illinois Institute of Technology

Eclipse Paho MQTT Client

Eclipse Paho MQTT Client

- An open-source MQTT client.
- Provide API for Java and many other languages.
  - Use Gradle to download the library and the source files automatically.
- An MQTT client is allowed to connect to the MQTT broker and serve both as a subscriber and publisher.

Managing the MQTT Client

```
public class Main implements AutoCloseable {
  ...
  public Main(SimConfig config) throws Exception {
    ...
    // start MQTT client
    this.mqtt = new MqttClient(config.getMqttBroker(),
      config.getMqttClientId(), new MemoryPersistence());
    this.mqtt.connect();
  }
  @Override
  public void close() throws Exception {
    http.close();
    mqtt.disconnect();
  }
  private final JHTTPhttp;
  private final MqttClient mqtt;
  ...
}
```

- Similar to how we manage the HTTP server.

MQTT Subscriber

- For our IoT simulator, we subscribe to MQTT topics to control the switches.
- This responsibility is similar to that of `HttpCommands`.
- Thus, let's implement `MqttCommands` similar to `HttpCommands`.

MqttCommands

```
public class MqttCommands {
  private final TreeMap<String, PlugSim> plugs =new TreeMap<>();
  private final String topicPrefix;

  public MqttCommands(List<PlugSim> plugs, String topicPrefix) {
    for (PlugSim plug: plugs)
      this.plugs.put(plug.getName(), plug);
    this.topicPrefix =topicPrefix;
  }
  ...
}
```

- We will need to store `topicPrefix` to be used later.

Handling Messages

```
public class MqttCommands {
  ...
  public String getTopic() {
    return topicPrefix+"/action/#";
  }
  public void handleMessage(String topic, MqttMessage msg) {
    try {
      logger.info("MqttCmd {}", topic);
      // switch on/off/toggle here
    }
    catch (Throwable th) {
      // Otherwise, Mqtt client will disconnect w/o error information
      logger.error("MqttCmd {}: {}", topic, th.getMessage(), th);
    }
  }
}
```

- Compute the topic wildcard to be subscribed and handle messages.
- Make sure to catch any `Throwable` like in `Runnable` – otherwise, the client may fail without printing any error information.

Subscribing to the Topic

```
public class Main implements AutoCloseable {
  ...
  public Main(SimConfig config) throws Exception {
    ...
    MqttCommands mqttCmd = new MqttCommands(plugs, config.getMqttTopicPrefix())
    logger.info("Mqtt subscribe to {}", mqttCmd.getTopic());
    this.mqtt.subscribe(mqttCmd.getTopic(), (topic, msg) -> {
      mqttCmd.handleMessage(topic, msg);
    });
  }
  ...
}
```

- Create the `MqttCommands` object and use a lambda expression to allow the MQTT client to use it.
  - No need to implement any interface for IoC.
- It is a good practice to encapsulate a complex lambda expression in a method.
- How to test `MqttCommands`?
  - It can be done by itself without using any MQTT client.

MQTT Publisher

```
public class PlugSim {
  ...
  public static interface Observer {
    void update(String name, String key, String value);
  }
  ...
}
```

- The state updates from the plugs need to be published to the MQTT broker.
  - As obtained from an `Observer`.
- Can you design something that can be tested without the MQTT client?

MqttUpdates

```
public class MqttUpdates {
  private  final String topicPrefix;
  public MqttUpdates(String topicPrefix) {
    this.topicPrefix =topicPrefix;
  }
  public String getTopic(String name, String key) {
    return  topicPrefix+"/update/"+name+"/"+key;
  }
  public MqttMessage getMessage(String value) {
    MqttMessage msg=new MqttMessage(value.getBytes());
    msg.setRetained(true);
    return  msg;
  }
}
```

- Just compute the topic and the message.
  - So that it can be tested just by itself.
- Use retained messages so other clients will be able to see the last updates.
- Similar to `MqttCommands`, `MqttUpdates` can be tested by itself without using any MQTT client.

Publishing the Updates

```
public class Main implements AutoCloseable {
  ...
  public Main(SimConfig config) throws Exception {
    ...
    MqttUpdates mqttUpd = new MqttUpdates(config.getMqttTopicPrefix());
    for (PlugSim plug: plugs) {
      plug.addObserver((name, key, value) -> {
        try {
          mqtt.publish(mqttUpd.getTopic(name, key), mqttUpd.getMessage(value));
        } catch (Exception e) {
          logger.error("fail to publish {} {} {}", name, key, value, e);
        }
      });
    }
  }
  ...
}
```

- Create the `MqttUpdates` object.
- Use lambda expression to create an `Observer` for each plug.
- For Project 3, you don't need to provide unit tests for `Main`.

Discussions

- Libraries supporting event-driven programming usually utilize one-method interfaces for IoC.
- It is quite convenient to use lambda expressions whenever one-method interfaces are needed, without the need to implement the interfaces.
- Pay attention to exceptions.
- Pay attention to the threads those interfaces/lambda expressions will be called.
  - Use locks when necessary.
  - Subscribed MQTT messages are handled in a thread managed by the MQTT client.
  - `Observers` are called and thus update messages are published from threads handling HTTP requests.