UNIVERSIDADE FEDERAL DE ITAJUBÁ - UNIFEI

Message Queuing Telemetry Transport

Equipe:

André Mury - 31010

André Arantes

Igor Gouveia

Lucas Mello

**I.** [**SUMÁRIO**](#_71gdormgm9w8) **2**

[**II. GLOSSÁRIO**](#_58ysbqy4sdgd) **3**

1. [Siglas](#_samtdcb0umt6) 3
2. [Ferramentas, aplicações e softwares](#_23hjhtlxk7n2) 3

1. [**Introdução**](#_v95zsdbdq4ix) **4**

[1.1. História:](#_qqqw24z6rrk0) 4

[1.2. O que é](#_gbhjacaiyvyj) 4

[1.3. Publicação / Assinatura:](#_e5qoain9x991) 4

[1.4. Estrutura:](#_gpzaf8wvu12d) 4

[1.5. Funcionamento:](#_93hyrhehjo9g) 5

[1.6. MQTT e IoT:](#_g21nsijpbxvx) 5

[1.7. HTTP e IoT:](#_hp3sc2um76d8) 5

2. [**Working Systems**](#_boqvry4xe3h6) **5**

[2.1. Como funciona o protocolo MQTT?](#_7u5en3v29ev5) 5

[2.2. *Broker*](#_i9b93pob5we) 6

[2.3. Por que o MQTT](#_jubgj8xlzd7u) 7

2.4. Por que não usar outros protocolos de rede 7

3. [**Tutorial**](#_dd3xbavl1p0p) **8**

[3.1. *Mosquitto*](#_kuof3c9yn0dt) 8

[3.2. *Maven*](#_ia51s1njoira) 9

[3.3. Iniciando seu primeiro projeto com MQTT](#_57qss64an9vt) 10

[3.3.1. *Eclipse Paho*](#_fd2exsoeirng) 11

[3.3.2. *Compiler Plugins*](#_nduyyogr6tmp) 12

3.4. Implementação 14

[3.4.1. *Publishe*](#_gdp2fxmxpebs)*r* 15

[3.4.2. *Subscriber*](#_nukkxie43emn) 17

[3.4.3. *Reply Getter*](#_gild9u8deor7) 19

[3.4.4. *Operation Controller*](#_478fk38lfken) 21

[3.4.5. *Calculator*](#_xtp4ziarezkf) 22

[3.5. *Compilation and Run Process*](#_s2dqpwf56dk3) 23

4. [**Conclusão**](#_dsn5e1bogwqg) **24**

# II. GLOSSÁRIO

## Siglas:

* 1. **MQTT** - *Message Queueing Telemetry Transport*
  2. **HTTP** - HyperText Transfer Protocol
  3. **IoT** - *Internet of Things*
  4. **IDE** - *Integrated Development Environment*
  5. **AMQP -** Advanced Message Queuing Protocol
  6. **XMPP -** Extensible Messaging and Presence Protocol
  7. **IM -** Instant Messaging

## Ferramentas, aplicações e softwares

* 1. ***Apache***- Empresa referência em desenvolvimento de *Frameworks* e ferramentas para desenvolvimento.
  2. ***Java*** *-* Linguagem proprietária da *Oracle/Sun*.
  3. ***Mosquitto*** *-* **MQTT** *broker.*
  4. ***Broker*** *-* entidade responsável por encaminhar mensagens publicadas em um canal a todos os inscritos neste independente de linguagem.
  5. ***Maven*** *- Plugin/Framework* gerenciador de dependências *Java.*
  6. ***Eclipse Paho* -** dependência proprietária *Eclipse* desenvolvida para gerenciar o protocolo **MQTT**.
  7. ***Callback***- método responsável por gerar uma resposta a um processamento.
  8. ***Payload***- parte do cabeçalho do protocolo que inclui toda a mensagem enviada a um tópico.
  9. ***Listening******State*** - estado em que o servidor ou máquina está escutando por mensagens em uma porta.
  10. ***Publisher*** - entidade que publica mensagens em um tópico.
  11. ***Subscriber*** - entidade que recebe as mensagens publicadas em um tópico.
  12. ***Server*** - entidade responsável por gerenciar todo o processamento de mensagens.
  13. ***Client***- entidade que consome o conteúdo dos servidores.

# Introdução

## 1.1. História:

O **MQTT** foi inventado e desenvolvido inicialmente pela *IBM* no final dos anos 90. Sua aplicação original era vincular sensores em pipelines de petróleo a satélites. No final de 2014, ele se tornou oficialmente um padrão aberto **OASIS**, com suporte nas linguagens de programação populares, usando diversas implementações de *software* livre.

## 1.2. O que é

O **MQTT** é um protocolo de ligação que especifica como os bytes de dados são organizados e transmitidos pela rede **TCP/IP**. Mas por motivos práticos, os desenvolvedores não precisam entender o protocolo de ligação. Basta saber que cada mensagem tem uma carga útil de comando e dados, usando um modelo de publicação e assinatura (*Publisher*/*Subscriber*) É um protocolo de rede leve e flexível que oferece o equilíbrio ideal para os desenvolvedores de **IoT**.

## 1.3. Publicação / Assinatura:

O protocolo **MQTT** define dois tipos de entidades na rede: um *message* *broker* e inúmeros clientes. O *broker* é um servidor que recebe todas as mensagens dos clientes e, em seguida, roteia essas mensagens para os clientes de destino relevantes. Um cliente é qualquer coisa que possa interagir com o broker e receber mensagens. Um cliente pode ser um sensor de IoT em campo ou um aplicativo em um data center que processa dados de **IoT**.

## 1.4. Estrutura:

Um servidor implementa estruturas de armazenamento de dados, referenciadas como tópicos. Cada tópico refere-se a uma estrutura de publicação (*publisher*) de dados, mas também possibilita a leitura (*subscriber*) dos dados. Um dos fatores mais importantes do protocolo **MQTT** são os tópicos, que identifica todas as mensagens.

O **MQTT** possui uma estrutura de pacotes de controle própria, que basicamente é constituída de três partes principais: cabeçalho fixo, cabeçalho variável e o *payload*.

1. Cabeçalho fixo -  composto de ao menos 2 *bytes*, possui 4 *bits* designados para definir o tipo do pacote de controle, 3 *bits* para *flags* e um *byte* adicional para suportar *bits* adicionais do cabeçalho variável e *payload*.

\* *Flags*: atualmente a maior parte dos parâmetros de *flag* (*CONNECT*, *CONNACK*) encontram-se reservados para usos futuros.

1. Cabeçalho variável - nem todos os tipos de mensagem **MQTT** possuem um cabeçalho variável, que basicamente traz informações da identificação do pacote.
2. *Payload -*  inclui a mensagem como um todo (normalmente os tópicos).

## 1.5. Funcionamento:

1. O cliente conecta-se ao *broker*. Ele pode assinar qualquer "tópico" de mensagem no broker. Essa conexão pode ser uma conexão **TCP/IP** simples ou uma conexão **TLS** criptografada para mensagens sensíveis.
2. O cliente publica as mensagens em um tópico, enviando a mensagem e o tópico ao *broker*.
3. Em seguida, o *broker* encaminha a mensagem a todos os clientes que assinam esse tópico.

## 1.6. **MQTT** e **IoT**:

1. Apenas leitura dos questionamentos.

## 1.7. **HTTP** e **IoT**:

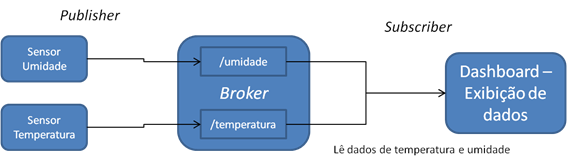
1. O **HTTP** é um protocolo síncrono. No mundo da **IoT**, a comunicação síncrona tem sido um problema devido ao grande número de dispositivos e à rede, muitas vezes não confiável e de alta latência.
2. **HTTP** é unidirecional. O cliente precisa iniciar a conexão. Em um aplicativo de **IoT**, os dispositivos e sensores geralmente são clientes, o que significa que eles não podem receber comandos da rede passivamente.
3. **HTTP** é um protocolo de um para um. O cliente faz uma solicitação e o servidor responde. É difícil e caro transmitir uma mensagem a todos os dispositivos na rede, o que é um caso de uso comum em aplicativos de **IoT**.
4. **HTTP** é um protocolo pesado com muitos cabeçalhos e regras. Ele não é adequado para redes restringidas.

# *Working Systems*

## 2.1. Como funciona o protocolo **MQTT**?

O princípio de funcionamento do protocolo parte da utilização de um Servidor **MQTT**, também conhecido como *broker* **MQTT**. Tal Servidor implementa estruturas de armazenamento de dados, referenciadas como tópicos. Cada tópico refere-se a uma estrutura de publicação (*publisher*) de dados, mas também possibilita a leitura (*subscriber*) dos dados.

Um dos fatores mais importantes do protocolo **MQTT** são os tópicos, que identifica todas as mensagens. O conceito é parecido com o de **URI**, com níveis separados por barras: os elementos da rede enviam diversos tópicos para o broker, e os subscribers escolhem quais os tópicos que querem subscrever. Vamos imaginar uma aplicação, em que se deseja monitorar as condições ambientais de um datacenter. Para isso temos das áreas de sensores: sensores de umidade e de temperatura, que nesse caso serão os nossos publishers. As informações coletadas serão armazenadas no *broker* sob os tópicos “/umidade” e “/temperatura”. Um terceiro nó, provavelmente pertencente a área de **NOC** da empresa, gostaria de ter essas informações, e, portanto, faz um subscriber para esses tópicos, como na imagem abaixo:



Ainda em cima do exemplo acima, imaginemos que no *datacenter* há duas áreas exemplo: 10 e 20, com diferentes sensores 5000, 5001, 4000, 4001 e que um departamento da empresa quer saber informações de temperatura da área 10. Um exemplo de mensagens trocadas, tanto dos *publishers*, quanto dos *subscribers* seriam os seguintes:

* *Publishers*



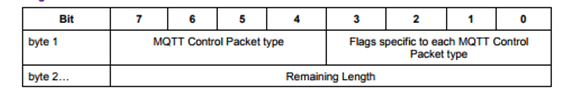
* *Subscribers*



O **MQTT** possui uma estrutura de pacotes de controle própria, que basicamente é constituída de três partes principais: Cabeçalho fixo, cabeçalho variável e o *payload*.

Principais tipos de pacotes: *CONNECT* (requisição de conexão ao servidor), *PUBLISH* (publicação de mensagem), *SUBSCRIBE* (subscrição).

*Flags*: atualmente a maior parte dos parâmetros de *flag* (*CONNECT*, *CONNACK*) encontram-se reservados para usos futuros. Apenas o tipo *PUBLISH* possui funções próprias e possui definições para entrega **DUP**, qualidade de serviço **QoS** e opções de retenção *RETAIN*.

*Payload*  —  inclui a mensagem como um todo (normalmente os tópicos). Os pacotes de confirmação (*CONNACK, PUBACK, SUBACK*) não necessitam de um *payload*.

## 2.2. *Broker*

O *broker* está no centro de qualquer protocolo de *PUBLISH*/*SUBSCRIBE*. Dependendo da implementação, um intermediário pode manipular até milhares de clientes **MQTT** conectados simultaneamente. O *broker* é responsável por receber todas as mensagens, filtrar as mensagens, determinar quem está inscrito em cada mensagem e enviar a mensagem para esses clientes inscritos. O *broker* também mantém as sessões de todos os clientes persistentes, incluindo assinaturas e mensagens perdidas. Outra responsabilidade do *broker* é a autenticação e autorização de clientes. Normalmente, o intermediário é extensível, o que facilita a autenticação, a autorização e a integração personalizadas nos sistemas de *back-end*. A integração é particularmente importante porque o intermediário é frequentemente o componente que é exposto diretamente na Internet, manipula muitos clientes e precisa passar mensagens para sistemas de análise e processamento a jusante. Em resumo, o intermediário é a central através do qual toda mensagem deve passar. Portanto, é importante que seu *broker* seja altamente escalável, integrável em sistemas *back-end*, fácil de monitorar e resistente a falhas.

## 2.3. Por que o MQTT

O **MQTT** é um protocolo de rede leve e flexível que oferece o equilíbrio ideal para os desenvolvedores de **IoT**:

* O protocolo leve permite a implementação em *hardware* de dispositivo altamente restringido e em redes de largura da banda limitada e de alta latência.
* Sua flexibilidade possibilita o suporte a diversos cenários de aplicativo para dispositivos e serviços de **IoT**.

## 2.4. Por que não usar algum dos outros inúmeros protocolos de rede

A maioria dos desenvolvedores já se acostumou aos serviços da *Web* **HTTP**. Então, por que não conectar os dispositivos de **IoT** aos serviços da *web*? O dispositivo poderia enviar seus dados como uma solicitação de **HTTP** e receber atualizações do sistema como uma resposta de **HTTP**. Esse padrão de solicitação e resposta tem algumas limitações graves:

* O **HTTP** é um protocolo síncrono. O cliente espera que o servidor responda. Os navegadores da *web* têm esse requisito, mas o custo é a baixa escalabilidade. No mundo da **IoT**, a comunicação síncrona tem sido um problema devido ao grande número de dispositivos e à rede, muitas vezes não confiável e de alta latência. Um protocolo de mensagem assíncrono é muito mais adequado para aplicativos de **IoT**. Os sensores podem enviar leituras e permitir que a rede descubra o caminho e a sincronização ideais para entregar aos dispositivos e serviços de destino.
* **HTTP** é unidirecional. O cliente precisa iniciar a conexão. Em um aplicativo de **IoT**, os dispositivos e sensores geralmente são clientes, o que significa que eles não podem receber comandos da rede passivamente.
* **HTTP** é um protocolo de um para um. O cliente faz uma solicitação e o servidor responde. É difícil e caro transmitir uma mensagem a todos os dispositivos na rede, o que é um caso de uso comum em aplicativos de **IoT**.
* **HTTP** é um protocolo pesado com muitos cabeçalhos e regras. Ele não é adequado para redes restringidas.

Pelos motivos citados acima, a maioria dos sistemas escaláveis de alto desempenho usam um barramento do sistema de mensagens assíncrono, em vez de serviços da web, para trocas de dados internas. Na verdade, o protocolo de sistema de mensagens mais popular que é usado em sistemas de middleware corporativos é chamado **AMQP** . No entanto, no ambiente de alto desempenho, a capacidade de computação e a latência da rede geralmente não são uma preocupação. O **AMQP** foi criado para assegurar a confiabilidade e a interoperabilidade em aplicativos corporativos. Ele possui um rico conjunto de recursos, mas não é adequado para aplicativos de **IoT** com restrição de recursos.

Além do **AMQP**, existem outros protocolos populares de sistema de mensagens. Por exemplo, o **XMPP** é um protocolo de mensagem instantânea (**IM**) ponto a ponto. Ele é pesado em recursos com suporte para casos de uso de **IM**, como presença e anexos de mídia. Em comparação com o **MQTT**, ele requer muito mais recursos no dispositivo e na rede.

# Tutorial

Neste tópico, nós iremos trabalhar na utilização do protocolo de telemetria para realizar uma tarefa simples, onde uma mensagem será enviada pela rede até um servidor, alguma operação será realizada e então um valor será retornado ao cliente e, para isso, necessitaremos implementar dois pequenos sistemas (client-server), portanto, vamos à configuração e ao processo de instalação. Após escolher a ***IDE*** que melhor se encaixa em suas necessidades, preciso lembrar que este processo será realizado em sistema Linux devido às suas vantagens como server.

A partir das premissas descritas na descrição do projeto, a escolha da linguagem para o desenvolvimento deste minicurso foi ***Java***, pois, é uma forma leve e sintática para trabalhar com o protocolo e existem diversos materiais disponíveis para estudo e capacitação.

## 3.1. *Mosquitto*

Para desenvolver um projeto a partir de um protocolo, utiliza-se um processo de middleware chamado *broker*, responsável pelas ações *publish-subscribe*, que retornam ao cliente apenas as informações solicitadas por ele intermediando a conexão direta com o provedor do serviço.

Antes de iniciar o projeto, precisaremos configurar este *broker*, para isso, com seu terminal aberto, digite:

### $ sudo apt-get update

### $ sudo apt-get install mosquitto

### $ sudo apt-get install mosquitto-clients

Este procedimento irá instalar a versão mais recente do broker utilizado, o *Mosquitto*, que será responsável pela função de *publish-subscribe* do projeto.

Para testar o funcionamento do *broker*, abra dois terminais, um para o *publisher* e outro para o *subscriber* e digite:

1. No primeiro:

### $ mosquitto\_sub -h 127.0.0.1 -t iot\_data

Este comando, fará com que o primeiro terminal fique em estado de espera (*listening*) para as mensagens que virão posteriormente.

1. No segundo:

### $ mosquitto\_pub -h 127.0.0.1 -t iot\_data -m "Hello world"

Este comando envia a mensagem para o **IP** 127.0.0.1 a partir do protocolo **MQTT**.

Observe no primeiro terminal, que a mensagem enviada pelo segundo foi recebida e publicada, se isso aconteceu, seu *broker* foi instalado corretamente e está preparado para o uso posterior na aplicação.

## 3.2. *Maven*

Como próximo passo, instalaremos o gerenciador de injeção de dependências para *Java*, o Maven, este que é proprietário (*open* *source*) da empresa *Apache*. O processo de instalação não é tão simples quando o *Mosquitto*, portanto, verifique se todos os passos foram seguidos corretamente:

1. Faça o download do arquivo no site oficial do *Maven*.

### <https://maven.apache.org/download.cgi>

1. Abra o terminal e mova-se ao diretório /*opt*:

### $ cd /opt

1. Extraia o .tar no diretório:

### $ sudo tar -xvzf /(**diretório-do-arquivo**)/apache-maven-**X.X.X**-bin.tar.gz

**\*Lembre-se de substituir “X.X.X” pela versão na qual foi baixada.**

1. Modifique a variáveis de ambiente do linux para poder utilizá-lo:
   1. Abra o arquivo de variáveis de ambiente:

$ sudo nano /etc/environment

* 1. Modifique a linha “**PATH**”, adicionando ao final:

### “:/opt/apache-maven-**X.X.X**/bin”, tornando-se semelhante à: PATH=”[conteúdo-anterior]**:/opt/apache-maven-X.X.X/bin**” **\***

* 1. Adicione ao final do documento, a linha:

### M2\_HOME=”/opt/apache-maven-**X.X.X”**

1. Atualize os comandos do *Maven*, digitando:

### sudo update-alternatives

### --install "/usr/bin/mvn" "mvn" "/opt/apache-maven-**X.X.X**/bin/mvn" 0 **\***

### sudo update-alternatives --set mvn /opt/apache-maven-**X.X.X**/bin/mvn

1. Complete a instalação do *Maven* em seu terminal, assim, pode-se usar a tecla *tab* para mostrar as opções do *plugin*:

### sudo wget <https://raw.github.com/dimaj/maven-bash-completion/master/>

### bash\_completion.bash --output-document /etc/bash\_completion.d/mvn

**\*Digite tudo em um único comando.**

1. Verifique a instalação do *Maven* digitando

### $ mvn --version

Se tudo ocorreu corretamente, sem indicação de erros e a ferramenta foi instalada corretamente, seu terminal deve retornar a versão do *Maven.*

## 3.3. Iniciando seu primeiro projeto com **MQTT**

O interessante de iniciar seu projeto com *Maven* é que podemos utilizar a **Injeção de Dependências**, dispensando da necessidade de instalar manualmente cada uma das ferramentas que serão utilizadas no projeto e, atualmente, o desenvolvimento de aplicações está caminhando para uma complexidade diferente dos tempos passados, pois, existe demasiada quantidade de *APIs* e *Frameworks* utilizados no desenvolvimento que contém dezenas ou centenas de pequenos outros componentes utilizados para realizar ações independentes umas das outras para concluir uma grande tarefa, chamados dependências.

Para criar seu primeiro projeto com *Maven* e **MQTT** é necessário, primeiramente, inicializar o projeto com a aplicação. Atento à isso, abra seu terminal e digite:

### 

### $ cd ~/diretório/de/projetos

### 

### $ mvn -B archetype:generate \

### -DarchetypeGroupId=org.apache.maven.archetypes \

### -DgroupId=com.mqtt.app \

### -DartifactId=**nome-da-pasta-projeto**

Espere o final do procedimento e note que o diretório escolhido será criado com uma estrutura básica para um projeto em Java.

Dentro da pasta do projeto, também foi criado um arquivo chamado *pom.xml*, o qual são definidas as configurações do *Maven* e é nele que indicaremos as dependências do projeto, portanto, abra este arquivo com um editor de texto ou *IDE*. Serão encontradas *tags* *xml* dentro deste, e, incluso, teremos a seção de dependências:

### <dependencies>

### <dependency>

### <groupId>junit</groupId>

### <artifactId>junit</artifactId>

### <version>3.8.1</version>

### <scope>test</scope>

### </dependency>

### **<< Este é o local onde colocaremos nossa primeira dependência >>**

### </dependencies>

Segundo o protocolo de injeção de dependências do *Maven*, adicionaremos nossa primeira dependência ao projeto.

## 3.3.1. *Eclipse Paho*

O projeto *Eclipse Paho* provê implementação open-source de protocolos **MQTT** e **MQTT-SN** destinado a aplicações novas, existentes e emergentes para a Internet das Coisas (**IoT**). [*ECLIPSE*.*ORG*]

Para adicionar a dependência, então, digite uma nova *tag* *<dependency>* no local indicado no exemplo:

### <dependencies>

### <dependency>

### <groupId>junit</groupId>

### <artifactId>junit</artifactId>

### <version>3.8.1</version>

### <scope>test</scope>

### </dependency>

### **<dependency>**

### **<groupId>org.eclipse.paho</groupId>**

### **<artifactId>org.eclipse.paho.client.mqttv3</artifactId>**

### **<version>1.2.0</version>**

### **</dependency>**

### </dependencies>

Após adicionar a dependência, é necessário também indicar onde esta está alocada, ie., seu repositório. Assim, dentro do arquivo *pom.xml*, logo acima da tag *<dependencies*>, indicaremos a tag <*repositories*>, com o endereço do repositório de sua dependência. Isso será responsável para indicar ao *Maven* e à sua **IDE**, de onde ela deve importar as classes, variáveis e métodos utilizados no projeto.

### **<repositories>**

### **<repository>**

### **<id>paho-mqtt-client</id>**

### **<name>Paho MQTT Client</name>**

### **<url>https://repo.eclipse.org/content/repositories/paho-releases/</url>**

### **</repository>**

### **</repositories>**

### <dependencies>

### <dependency> ...

Feito isso, agora adicionaremos nossos *plugins* para compilar o projeto corretamente.

## 3.3.2. *Compiler Plugins*

Se tentar executar o projeto com estes plugins a partir do ambiente de desenvolvimento, provavelmente teremos êxito, pois, a **IDE** é quem se preocupa em importar todas as classes e definir os *classpaths* para o projeto, porém, ao construir o projeto (*build*) aparecerão alguns problemas, pois o *Maven* irá compilar apenas o projeto sem as dependências, por isso o próprio *Maven* nos proporciona alguns *plugins* para importar estas dependências para dentro do arquivo .*jar* gerado ao final da *build*.

Para adicionar *plugins* ao seu projeto *Maven*, utilizaremos a tag <*plugins>* na construção do projeto (<*build*>), portanto, teremos nossos plugins dentro dessas *tags*:

### 

### </dependencies>

### **<build>**

### **<plugins>**

### **Seus *plugins***

### **</plugins>**

### **</build>**

Os *plugins* utilizados e suas importações são:

1. *Maven Shade -* Compila as dependências junto ao projeto

### <plugin>

### <groupId>org.apache.maven.plugins</groupId>

### <artifactId>maven-shade-plugin</artifactId>

### <version>1.6</version>

### <executions>

### <execution>

### <phase>package</phase>

### <goals>

### <goal>shade</goal>

### </goals>

### </execution>

### </executions>

### </plugin>

1. *Maven Assembly -* Monta o projeto com dependências

### <plugin>

### <artifactId>maven-assembly-plugin</artifactId>

### <configuration>

### <archive>

### <manifest>

### <mainClass>com.mqtt.app.App</mainClass>

### </manifest>

### </archive>

### <descriptorRefs>

### <descriptorRef>jar-with-dependencies</descriptorRef>

### </descriptorRefs>

### </configuration>

### </plugin>

1. *Maven Jar* - Realiza o build do projeto em .*jar* com *classpaths*

### <plugin>

### <groupId>org.apache.maven.plugins</groupId>

### <artifactId>maven-jar-plugin</artifactId>

### <version>2.4</version>

### <configuration>

### <archive>

### <manifest>

### <mainClass>com.mqtt.app.App</mainClass>

### <addClasspath>true</addClasspath>

### </manifest>

### </archive>

### </configuration>

### </plugin>

Seu arquivo *pom.xml* deve estar listando, neste momento, o repositório *Paho*, o próprio *Eclipse Paho MQTTv3*, e estes três *plugins* do *Maven*, com a seguinte estrutura:

### <project ...>

### <modelVersion></modelVersion>

### <groupId></groupId>

### <artifactId></artifactId>

### <packaging></packaging>

### <version></version>

### <name></name>

### <url></url>

### <repositories>

### ...

### </repositories>

### <dependencies>

### <dependency> *(Dependency 1..)* </dependency>

### ...

### </dependencies>

### <build>

### <plugins>

### </plugin> *(Plugin 1…)* </plugin>

### …

### </plugins>

### </build>

### </project>

Feito isso, salve seu arquivo e nosso ambiente está pronto para começar a implementação.

3.4. Implementação

Agora que já temos nosso projeto *Maven* em *Java* com suas dependências e *plugins* configurados, estamos prontos para começar a implementação. O primeiro passo é definir a estrutura de nosso projeto, portanto, vamos a ela. Teremos diversos pacotes para suprir e organizar nosso projeto, são eles: ***app, app.Controllers, app.Models, app.Publish, app.Subscribe, app.States***.

1. *app -* este é o pacote principal, onde ficará o arquivo ***app.java***, fazendo papel de *main class*. *Maven* já cria este arquivo por padrão e o arquivo de configurações ***Config***.
2. *app.Controllers* - este é o pacote responsável por guardar nossos controladores. Dentro deste, crie uma *Java Class* ***OperationController***.
3. *app.Models -* este pacote residirá nossos modelos, as classes que realizarão operações de processamento. Insira a classe ***Calculator***.
4. *app.Publish* **-** aqui ficarão as classes responsáveis pela publicação, incluindo seu *reply subscriber.* Crie as classes: ***ReplyGetter*** e ***PublisherCallback***.
5. *app.Subscribe* - este pacote residirá as classes responsáveis pelo serviço de *subscriber*. Em nosso caso, simularemos um servidor enviando uma resposta para a requisição do *publisher*. Crie as classes: ***Subscriber*** e ***SubscriberCallback.***
6. *app.States* - este pacote residirá o estado de *publishing*, pois, é um estado imutável acessado pelas duas partes. Crie a classe: ***Publisher***.

\*Note que, os dois estados (*publisher* e *subscriber*) residem no mesmo projeto, porém são desconexos, ie., não possuem interação entre si. A classe *Operation Controller* só é acessada pelo *Subscriber Callback*.

Temos então a seguinte hierarquia:

* ***com.mqtt.app***
  + *App*
  + *Config*
* ***com.mqtt.app.Controller***
  + *OperationController*
* ***com.mqtt.app.Models***
  + *Calculator*
* ***com.mqtt.app.Publish***
  + *PublisherCallback*
  + *ReplyGetter*
* ***com.mqtt.app.Subscribe***
  + *Subscriber*
  + *SubscriberCallback*
* ***com.mqtt.app.states***
  + *Publisher*

Antes de prosseguir com a implementação, devemos escrever o arquivo de configurações, portanto, abra-o e insira o código a seguir:

### public class Config {

### 

### static final String DEFAULT\_TOPIC = "iot\_data"; //Tópico de subscribe

### static final int DEFAULT\_TIMEOUT = 100;

### static final String DEFAULT\_PROTOCOL = "tcp";

### static final String DEFAULT\_HOST = "ip.do.*broker*"; //IP da máquina com *Broker*.

### static final String DEFAULT\_PORT = "1883" //Porta padrão do broker *Mosquitto*;

### static String topic = DEFAULT\_TOPIC;

### static String connProtocol = DEFAULT\_PROTOCOL;

### static String host = DEFAULT\_HOST;

### static String port = DEFAULT\_PORT;

### static int timeout = DEFAULT\_TIMEOUT;

### [Insira aqui todos os getters para as variáveis de configuração, **exceto** para DEFAULT\_\*]

### public static String getFullURI() {

### return getConnProtocol() + "://" + getHost() + ":" + getPort();

### }

### }

\*Esta classe não possui construtor.

## 3.4.1 *Publisher*

Primeiramente, implementaremos o Publisher, portanto, abra o arquivo *Publisher.java* e insira o código a seguir. Não se preocupe com as classes que ainda não foram implementadas nos *imports*, insira-as assim mesmo, pois trataremos disso adiante.

### import com.mqtt.app.Config; import com.mqtt.app.Publish.ReplyGetter;

### import org.eclipse.paho.client.mqttv3.MqttClient;

### import org.eclipse.paho.client.mqttv3.MqttException;

### import org.eclipse.paho.client.mqttv3.MqttMessage;

### 

### public class Publisher{

### MqttClient client;

### MqttMessage msg;

### String serverURI;

### String clientId;

### public Publisher() throws MqttException {

### serverURI = Config.getFullURI();

### clientId = MqttClient.generateClientId();

### this.client = new MqttClient(serverURI, clientId);

### this.msg = new MqttMessage();

### }

### public boolean publish(String message, String topic) throws MqttException {

### try {

### msg.setPayload((this.clientId + ": " + message).getBytes());

### client.publish(topic, this.msg);

### return true;

### } catch (MqttException e) {

### System.out.println(e);

### }

### return false;

### }

### public boolean connect() throws MqttException {

### try {

### this.client.connect();

### return true;

### } catch (MqttException e) {

### System.out.println(e);

### }

### return false;

### }

### public boolean disconnect() throws MqttException {

### try {

### this.client.disconnect();

### System.out.println("Client disconnected.");

### return true;

### } catch (MqttException e) {

### System.out.println(e);

### }

### return false;

### }

### /\*

### \*\*Este método ainda não tem funcionalidade, implementaremos a classe \*\*ReplyGetter mais \*\* adiante. \*/

### public static boolean getReply() throws MqttException, InterruptedException {

### ReplyGetter psub = new ReplyGetter();

### if (psub.connect("**reply-topic**")) {

### return true;

### }

### return false;

### }

### }

Esta é nossa classe que realizará o trabalho de publicar no tópico. Não trataremos erros de queda de conexão ou falta de resposta em nenhuma das classes, visto que este é um projeto de exemplo simples.

Agora, temos de implementar um *Callback* para o publicador, portanto, no arquivo de classe *PublisherCallback*, insira o código a seguir:

### import org.eclipse.paho.client.mqttv3.IMqttDeliveryToken;

### import org.eclipse.paho.client.mqttv3.MqttCallback;

### import org.eclipse.paho.client.mqttv3.MqttMessage;

### public class PublisherCallback implements MqttCallback {

### 

### public void connectionLost(Throwable thrwbl) {

### System.out.println("Connection lost.");

### }

### 

### //Este método retorna o estado de que uma mensagem chegou.

### public void messageArrived(String string, MqttMessage mm) throws Exception {

### string = new String(mm.getPayload());

### System.out.println("Reply: " + unjoin(string, ": ")[1] + ".");

### ReplyGetter.unsubscribe();

### }

### 

### public void deliveryComplete(IMqttDeliveryToken imdt) {

### //este método não será utilizado nesse exemplo.

### }

### 

### //Este método separa a *String payload* em um array de acordo com o delimitador.

### private String[] unjoin(String payload, String delimiter) {

### return payload.split(delimiter);

### }

### }

## 3.4.2. Subscriber

A classe subscriber será responsável por subscrever em algum tópico que, neste caso, será o mesmo tópico da classe publisher, definido em *Config.DEFAULT\_TOPIC*.

Insira no arquivo *Subscriber*, o seguinte código:

### public class Subscriber {

### 

### static MqttClient client;

### private String serverURI;

### public Subscriber() throws MqttException {

### this.serverURI = Config.getFullURI();

### client = new MqttClient(serverURI, MqttClient.generateClientId());

### }

### public static boolean connect(String topic) throws MqttException {

### try {

### client.setCallback(new SubscriberCallback());

### client.connect();

### System.out.println("Subscribed to topic " + topic + ".");

### client.subscribe(topic);

### return true;

### } catch (MqttException e) {

### System.out.println(e);

### }

### return false;

### }

### 

### public boolean disconnect() {

### try {

### client.disconnect();

### return true;

### } catch (MqttException e) {

### System.out.println(e);

### }

### return false;

### }

### 

### }

Da mesma forma que implementamos um *PublisherCallback*, implementamos também um *callback* para o *subscriber.* As classes de *Callback* são responsáveis por gerar algum tipo de resposta quando algum evento predefinido acontecer. Em nosso caso, os eventos são mensagens que chegam até o tópico os quais subscrevemos, tanto no caso do *Publisher* quando no *Subscriber*.

### import com.mqtt.app.Config;

### import com.mqtt.app.Controller.OperationController;

### import org.eclipse.paho.client.mqttv3.IMqttDeliveryToken;

### import org.eclipse.paho.client.mqttv3.MqttCallback;

### import org.eclipse.paho.client.mqttv3.MqttException;

### import org.eclipse.paho.client.mqttv3.MqttMessage;

### public class SubscriberCallback implements MqttCallback {

### 

### public void connectionLost(Throwable thrwbl) {

### System.out.println("Connection lost. Trying to reconnect..");

### try {

### reconnect();

### } catch (MqttException ex) {

### System.out.println("Connection has dropped. Please restart the”

### +” subscriber module.");

### }

### }

### 

### /\*

### \*\*Este método é responsável por gerar uma resposta a um tópico a partir do \*\*Operation controller para um canal específico de ***reply*** definido tanto no \*\*publisher quanto no subscriber, simulando assim, uma arquitetura \*\**client-server*.

### \*/

### public void messageArrived(String string, MqttMessage mm) throws Exception {

### string = new String(mm.getPayload());

### OperationController opc = new OperationController(string);

### opc.init();

### }

### public void deliveryComplete(IMqttDeliveryToken imdt) {

### //este método não será utilizado nesse exemplo.

### }

### private boolean reconnect() throws MqttException {

### try {

### Subscriber.connect(Config.getTopic());

### return true;

### } catch (MqttException e) {

### System.out.println(e);

### }

### return false;

### }

### }

Ao fim, nossa classe de *callback* e o estado de *subscriber* estão completos. Partiremos às duas últimas classes que permitirão nosso serviço funcionar: *OperationController* e *ReplyGetter.*

## 3.4.3 *Reply Getter*

Este é o procedimento o qual permitirá nosso programa receber uma resposta de um canal simulado como servidor. Esta classe irá, basicamente, se subscrever em um canal específico de *reply*, receber a resposta e desinscrever. O procedimento responsável por controlar estas funções é a classe *OperationController*, portanto, não é possível executar uma sem a outra.

No arquivo da classe, insira o seguinte código:

### import com.mqtt.app.Config;

### import org.eclipse.paho.client.mqttv3.MqttClient;

### import org.eclipse.paho.client.mqttv3.MqttException;

### 

### public class ReplyGetter {

### 

### static MqttClient client;

### private String serverURI;

### private static String t;

### public ReplyGetter() throws MqttException {

### this.serverURI = Config.getFullURI();

### client = new MqttClient(serverURI, MqttClient.generateClientId());

### }

### 

### public boolean connect(String topic) throws MqttException, InterruptedException

### {

### t = topic;

### try {

### client.setCallback(new PublisherCallback());

### client.connect();

### System.out.println("Subscribed to topic " + topic + ".");

### client.subscribe(t);

### 

### return true;

### } catch (MqttException e) {

### System.out.println(e);

### }

### return false;

### }

### public boolean disconnect() {

### try {

### client.disconnect();

### return true;

### } catch (MqttException e) {

### System.out.println(e);

### }

### return false;

### }

### 

### public static boolean unsubscribe() throws MqttException {

### try {

### client.unsubscribe(t);

### return true;

### } catch (Exception e) {

### System.out.println(e);

### }

### return false;

### }

### }

### 

Como pode ser notado, esta classe funciona como um *pseudo-subscriber* que se conecta rapidamente a um tópico e fecha em seguida. É necessário separar este do *subscriber* pois se tratam de dois projetos diferentes que não pode se interferir. O ideal seria criar em projetos *java* diferentes, mas não há necessidade aqui.

## 3.4.4. *Operation Controller*

Esta classe será a responsável por solicitar outros procedimentos que serão realizados pela simulação de servidor.

No arquivo referente à classe, adicione:

### import com.mqtt.app.Models.Calculator;

### import com.mqtt.app.states.Publisher;

### import org.eclipse.paho.client.mqttv3.MqttException;

### 

### public class OperationController {

### 

### String args;

### Publisher pub;

### 

### public OperationController(String args) {

### this.args = args;

### }

### 

### public void init() throws MqttException {

### try {

### pub = new Publisher();

### } catch (MqttException e) {

### System.out.println(e);

### }

### System.out.println("Operation Controller successfully initialized.");

### doOperation(args);

### }

### private String getClientId(String message) {

### return this.unjoin(message, ": ")[0];

### }

### private String[] getValues(String payload) {

### return this.unjoin(this.unjoin(payload, ": ")[1], " ");

### }

### 

### private String[] unjoin(String payload, String delimiter) {

### return payload.split(delimiter);

### }

### 

### public void doOperation(String payload) throws MqttException {

### String clientId = this.getClientId(payload);

### String[] arr = this.getValues(payload);

### int reply;

### 

### Calculator calc = new Calculator();

### 

### reply = calc.sumValues(arr);

### 

### if (callBack(Integer.toString(reply), clientId)) {

### System.out.println("Reply sent.");

### } else {

### System.out.println("Reply failed.");

### }

### }

### 

### private boolean callBack(String message, String clientId) {

### String topic = "**reply-topic**";

### 

### try {

### if (pub.connect()) {

### this.pub.publish(message, topic);

### pub.disconnect();

### return true;

### }

### } catch (MqttException e) {

### System.out.println(e);

### }

### return false;

### }

### }

### 

Agora nosso sistema está quase completo: envia requisições por *publish* e envia respostas a partir do subscriber, porém, note o uso da classe *Calculator* que ainda não está implementada. Este é nosso último passo. A classe em questão é um exemplo, neste local, podem ser definidas diversas opções para encaminhar a diversos processamentos diferentes, à escolha do desenvolvedor.

## 3.4.5 - *Calculator*

Nosso objetivo nesse projeto é simplesmente realizar uma requisição-resposta simples via protocolo **MQTT** e isso já conseguimos. Agora vamos ao último passo que é a implementação de uma simples calculadora que soma os inteiros à ela passados. Na classe *Calculator*, insira o código a seguir:

### public class Calculator {

### 

### public int sumValues(String[] arr) {

### int result = 0;

### System.out.print("Sum: ");

### System.out.print(String.join(" + ", arr));

### for (String a : arr) {

### result += Integer.parseInt(a);

### }

### System.out.println(" Done.");

### System.out.println("Result: " + s);

### return result;

### }

### }

Esta classe tem a função *sumValues()* que simplesmente soma indefinidos números em uma *array string* encaminhada do controlador de operação.

## 3.5. *Compilation and Run Process*

Se as configurações *pom.xml* estiverem sido feitas corretamente, não haverão problemas neste procedimento. Abra seu terminal.

Para garantir a completude das dependências e *plugins*, digite em seu terminal:

### $ mvn exec:java -D exec.mainClass=**com.mqtt.app**.App

\* Substitua **com.mqtt.app** para o nome correto do pacote contendo sua classe *App*.

Se tudo ocorrer corretamente, sua aplicação será executada. *Ctrl+c* para fechá-la. Agora, digite o seguinte comando em seu terminal para compilar seu aplicativo e executá lo em qualquer máquina equipada com *Java* *Environment*.

### $ cd diretorio/do/projeto

### $ mvn clean install package

O processo de *packaging* irá compilar o projeto e todas as suas dependências dentro da pasta criada *target*, na pasta de seu projeto e nela, um arquivo .*jar* com o nome definido em *pom.xml*. Para executá-la, digite em dois terminais diferentes:

### $ cd diretorio/do/projeto

### $ chmod +x target/nome-da-aplicacao.jar

### $ java -jar “target/nome-da-aplicacao.jar”

Em um deles, digite *pub* e em outro, *sub.* O aplicativo instanciará o projeto correto permitindo a troca de mensagens pela rede. Caso dois computadores estejam em uma rede local e a máquina que possui o *broker Mosquitto* instalado (pode ser uma outra máquina na rede local, que não esteja executando o aplicativo) estiver em funcionamento e com a porta 1883 escutando, pode-se executar um *subscriber* em uma delas e o *publisher* em outra.

A nossa função de calculadora interpreta uma array de numeros separados por espaço, portanto, digite no terminal uma sequência de números separada por um espaço, tal como o exemplo abaixo e note tanto no *subscriber* quanto no *publisher*, os *callbacks* gerados.

### 

|  |  |
| --- | --- |
| *Subscriber state* | *Publisher state* |
| Welcolme to the MQTT Server-client Project!What do you want to do?Publish or Subscribe: | Welcolme to the MQTT Server-client Project!What do you want to do?Publish or Subscribe: |
| sub | pub |
| Subscribed to topic iot\_data.Messages arrived: | What do you want to send? |
| *Esperando mensagem...* | 1 3 5 2 99 |
| *Esperando mensagem...* | Message sent. |
| Operation Controller successfully initialized.Sum: 1 + 3 + 5 + 2 + 99 Done.Result: 110Client disconnected.Reply sent. | *Esperando resposta...* |
| *Esperando mensagem...* | Reply: 110. |
| *Esperando mensagem...* | What do you want to send? |
| ... | ... |

### 

A partir deste ponto, segue da sua criatividade para elaborar métodos mais complexos para a aplicação.

## 4. Conclusão

Este trabalho tratou o protocolo **MQTT**, abordando a sua história e mostrando que sua aplicação inicial era a comunicação entre tubos de transporte de petróleo e satélites. Com o passar do tempo, este se difundiu, muito se deve ao fato de ser simples o suficiente sem deixar de abranger uma série de características como segurança, qualidade de serviço e facilidade de implementação.

Nasceu visando usar a infraestrutura e se integrar com o protocolo **TCP/IP**, sendo projetado para aplicações que usam pouca banda de rede, pois, ao invés de utilizar o tradicional padrão *request-response*, é utilizado *publisher-subscribe*, o que dá um maior dinamismo ao transporte de dados. Um ponto que causou um grande impacto no desenvolvimento e aceitação do **MQTT** é a sua flexibilidade, permitindo o suporte a diversos cenários de aplicações para dispositivos e serviços de **IoT**. Os protocolos anteriores ao **MQTT** não possuíam a flexibilidade que ele proporciona, e esta é uma necessidade da **IoT**. O **HTTP** pode ser usado como exemplo, por ser um protocolo síncrono, unidirecional e pesado, pelo fato de possuir cabeçalhos complexos e muitas regras.