POLITÉCNICO DE LEIRIA
ESCOLA SUPERIOR DE TECNOLOGIA E GESTÃO

DEPARTAMENTO DE ENGENHARIA INFORMÁTICA

www.dei.estg.ipleiria.pt

Systems Integration

Messaging Systems

Topics:
- Background
- Messaging systems
- Web Services or Messaging?
- Mosquitto Messaging Broker
- Examples and exercises

©2021-2022: {nuno.costa, marisa.maximiano}@ipleiria.pt

Background

There is no optimal solution that can be applied to all integration problems, which means that, different integration problems require different solutions or approaches (beyond Web Services).

Web Services (studied in previous classes) are part from a set of technologies that support systems or applications integration by direct connection or point-to-point, which means that Web Services are interesting to connect a reduced number of systems, otherwise the integration solution will become too much complex, hard to maintain and without scalability. Take this example:

Suppose a company or institution where 3 heterogeneous applications are used (these applications do not interoperate among themselves). Now, imagine that the company has plans to integrate the 3 applications to get a unified system that mimics the behaviour of a single system.

If Web Services are used, 3 connections are expected to connect all the existing applications, which is not a big deal. However, if the number of applications reach the 10 units, 45 connections must be established and handled, according to the formula **N(N-1)/2**, where N represents the number of applications that must interoperate.

On the one hand, Web Services follow the request/response model and the server, and the client must be active at the same time and the server must have enough resources to accept a bunch of clients (there is no contention by default). On the other hand, if the server goes offline, the client is responsible for handling all errors that come from that situation and if it is the client that goes down then the response is lost.

The Web Services are very interesting for situations where the client expects an immediate response from the server (**synchronous**).

1. Messaging systems

Communication based on message passing is a style where one or more queues are used between message sender and message receiver, where both ends can put or retrieve messages from the queues hence following a very flexible and "**disconnected**" way of communication.

A remarkable characteristic of messaging systems is the ability for communication when the ends may not be active at the same time, once the messages are stored in the queues (or even persisted in the hard disc) until proper delivery. Besides this, this approach naturally implements contention, avoiding systems saturation in situations of high load. In this way, neither the sender nor the receiver has to worry about network communications or offline errors because the messaging system handles that in a transparent way.

Once messaging systems are **asynchronous** by nature (but also support the synchronous approach) they are not very suitable for synchronous requests. Additionally, as messaging systems do not rely in point-to-point connections among applications or systems, in a 3 applications scenario only 3 connections are requested and in a 10 applications scenario only 10 connections are required.

2. Web Services or Messaging?

As already stated before, there is no unique solution for all integration problems. While Web Services are tailored for synchronous interactions or for scenarios with a small number of applications, messaging is tailored for scenarios where asynchronous interactions are required or for scenarios where lots of applications need to interact among themselves (e.g. Facebook, Twitter, etc.). Next, some tips are presented to identify the better approach for systems or applications integration:

- Do you want to handle the network errors or let them to the integration middleware? Messaging systems are asynchronous and can retry operations in the presence of failures.
- Choose a messaging system if clients do not need to actively wait for response.
- Messaging systems are tailored for long running requests once client will not be blocked waiting for the response.
- Messaging systems turn the integration solution high scalable once high request loads are naturally serialized into the processing queues and processing is naturally distributed through the time.
- Web services and messaging systems are not mutually exclusives. It is possible to create a messaging system where the send and receive interface is implemented using Web Services.

Note: Despite messaging systems appearing to solve the problems of Web Service, messaging systems have also problems such as message format or the communication protocol.

Enterprise Service Buses (ESB), which are based on messaging, have additional functionalities such as data transformations, support for multiple communication protocols and even business support logic to address the limitations of the messaging systems. BizTalk is an example of an ESB.

3. Mosquitto Messaging Broker

Eclipse Mosquitto™ [mosquitto.org] is an open source (EPL/EDL licensed) message broker that implements the MQTT protocol [mqtt.org/] versions 3.1 and 3.1.1. and can be used to connect different systems or applications following the publish/subscribe model. Alternatives to Mosquitto can be RabbitMQ, HiveMQ, Kafka, ActiveMQ, etc.

3.1. The Publish/Subscribe model

The publish/subscribe model defines 3 roles: the publisher, the subscriber and the middleman called broker. **Publishers**, as the name says, are the entities that generate and publish events or information, while **subscribers** are the entities that are interested in some type of events or information (they are interested in the work of publishers). By its turn, the **broker** is the entity that mediates communication or connections between publishers and subscribers. In this perspective, the publishers (producers) don't care about the number of subscribers (consumers) nor the time the subscribers are active.

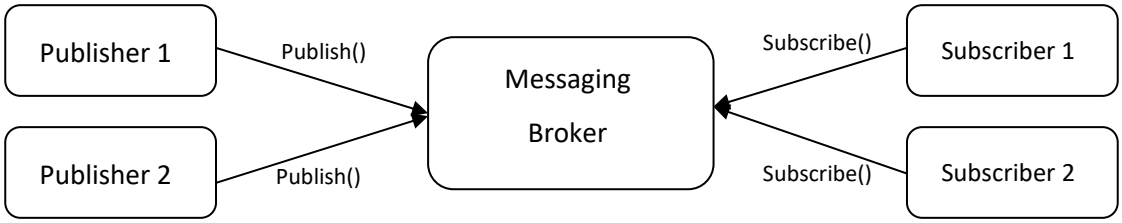

Fig. 1 – Publish/subscribe model

More concretely, the messaging broker creates the **topic** (e.g. channel) abstraction to be possible to handle multiple independent flows of information and to optimize the communication. This allows subscribers to only listen to communication channels (e.g. topics) they are interested to. Apart from permissions and security settings, one publisher can publish in one or more topics and one subscriber can subscribe to one or more topics. Figure 2 presents the publish/subscribe model but now with more detail.

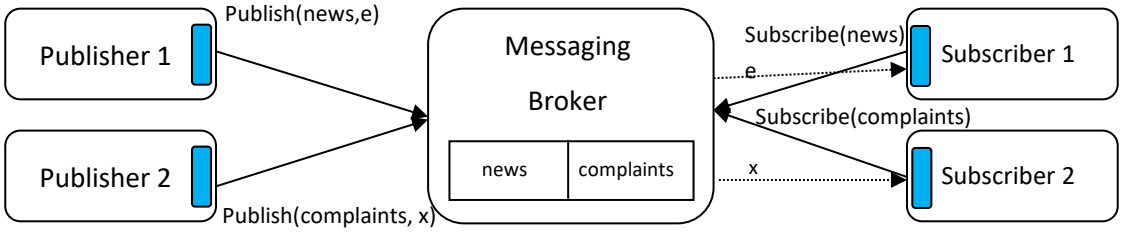

Fig. 2 – Publish/subscribe model with topics and access middleware

The blue rectangles represent the middleware (API) to access the messaging broker and this middleware plays a very important role in the communications reliability because, usually, that middleware implements the send & forget, plus Store and Forward approaches. This means that, when a publisher calls the **publish()** function it does not care about any kind of errors (e.g. communication) because the send & forget approach is implemented by the local middleware. On the other hand, the store and forward means that all in-transit messages are stored in order to survive to system downs, system crashes or other kind of system sever problems. In the subscriber's side, the middleware abstracts all the communications and deliver messages (asynchronously) to the subscriber application. Depending on configuration settings, a new subscriber can receive all past messages of a specific topic without any extra line of code.

3.2. The MQTT Protocol

MQTT is a machine-to-machine (M2M)/"Internet of Things" connectivity protocol. It was designed as an extremely lightweight publish/subscribe messaging transport. It is useful for connections with remote locations where a small code footprint is required and/or network bandwidth is at a premium. For example, it has been used in sensors communicating to a broker via satellite link, over occasional dial-up connections with healthcare providers, and in a range of home automation and small device scenarios. It is also ideal for mobile applications because of its small size, low power usage, minimized data packets, and efficient distribution of information to one or many receivers [from mqtt.org]

By default, the MQTT protocol uses the TCP/IP port 1883 (reserved with IANA) and port 8883 for using MQTT over SSL. MQTT is in the process of undergoing standardization at OASIS.

In terms of security, MQTT allows the usage of login and password with the MQTT packet and SSL if the SSL impact is acceptable to the system. Additional security could be added by encrypting data at application layer (not included in the protocol).

3.3. A word about QoS

MQTT defines three levels of QoS that could be assigned to each message being published: "at most once", "at least once" and "exactly once". Do not forget that higher QoS levels consume more resources than the lower ones. This could be important if you are dealing with resource poor hardware (e.g. sensor nodes).

3.4. Client libraries (and .Net)

MQTT, in this case Mosquitto, has lots of client APIs that cover the most known programming languages and systems such as Arduino, Bash, C, C++, Delphi, Erlang, Java, Javascript, Node.js, .NET, Objective-C, Perl, PHP, Python, Ruby, Qt, Swift, etc. See https://github.com/mqtt/mqtt.github.io/wiki/libraries for more details.

For this worksheet, we will use the .NET API (https://mosquitto.org/download) which has dependences from OpenSSL and PThreads (handled during installation).

To find out if Mosquitto is ready to be used, execute the following commands in separated consoles:

```
mosquitto_sub -v -t 'news'                    (to subscribe news topic/channel)
mosquitto_pub -t 'news' -m "Hello World!"     (to post message to topic)
```

To integrate Mosquitto .NET client library into a C# project do the following:

- Open Visual Studio and create a sample (form based) application:
- Now, run the following command in the Package Manager Console/or Nuget:
- Install-Package M2Mqtt
- The Project has now support for Mosquitto/MQTT.

4. Examples and exercises

4.1. Example 1 – Single app that publishes and subscribes

Example 1 shows a single (form based) application that uses the same connection to publish and to subscribe to two topics: news and complaints. Note that the message broker is in the local computer (e.g. 127.0.0.1).

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Linq;
using System.Net;
using System.Text;
using System.Threading.Tasks;
using System.Windows.Forms;
using uPLibrary.Networking.M2Mqtt;
using uPLibrary.Networking.M2Mqtt.Messages;

namespace Mosquitto1
{
    public partial class Form1 : Form
    {
        MqttClient mClient = new MqttClient(IPAddress.Parse("127.0.0.1")); //OR use the broker hostname
        string[] mStrTopicsInfo = { "news", "complaints" };

        public Form1()
        {
            InitializeComponent();
        }


        private void Form1_Load(object sender, EventArgs e)
        {
            mClient.Connect(Guid.NewGuid().ToString());
            if (!mClient.IsConnected)
            {
                MessageBox.Show("Error connecting to message broker...");
                return;
            }

            //Specify events we are interest on
            //New Msg Arrived
            mClient.MqttMsgPublishReceived += client_MqttMsgPublishReceived;
            //This client's subscription operation id done
            //mClient.MqttMsgSubscribed += client_MqttMsgSubscribed;
            //This client's unsubscription operation is done
            //mClient.MqttMsgUnsubscribed += client_MqttMsgUnsubscribed;

            //Subscribe to topics
            byte[] qosLevels = { MqttMsgBase.QOS_LEVEL_EXACTLY_ONCE,
                                 MqttMsgBase.QOS_LEVEL_EXACTLY_ONCE};//QoS
            mClient.Subscribe(mStrTopicsInfo, qosLevels);
        }

        void client_MqttMsgPublishReceived(object sender, MqttMsgPublishEventArgs e)
        {
            MessageBox.Show("Received = " + Encoding.UTF8.GetString(e.Message) + " on topic " +
                            e.Topic);
        }

        /*void client_MqttMsgUnsubscribed(object sender, MqttMsgUnsubscribedEventArgs e)
        {
            MessageBox.Show("UNSUBSCRIBED WITH SUCCESS");
        }*/

        /*void client_MqttMsgSubscribed(object sender, MqttMsgSubscribedEventArgs e)
        {
            MessageBox.Show("SUBSCRIBED WITH SUCCESS");
```

```
        }*/

        private void OnButtonPublish_Click(object sender, EventArgs e)
        {
            if (mClient.IsConnected)
            {
                mClient.Publish("news", Encoding.UTF8.GetBytes("Hello World!"));
            }
        }

        private void Form1_FormClosing(object sender, FormClosingEventArgs e)
        {
            if (mClient.IsConnected)
            {
                mClient.Unsubscribe(mStrTopicsInfo); //Put this in a button to see notif!
                mClient.Disconnect(); //Free process and process's resources
            }
        }
    }
}
```

4.1.1. Exercise

Change the above implementation to show the topics in the *combobox* when the form is loading. The user can put the message to be shown in the *textbox* (set textbox multiline property to true). Add an unsubscribe button to the form that allows the user to unsubscribe all topics.

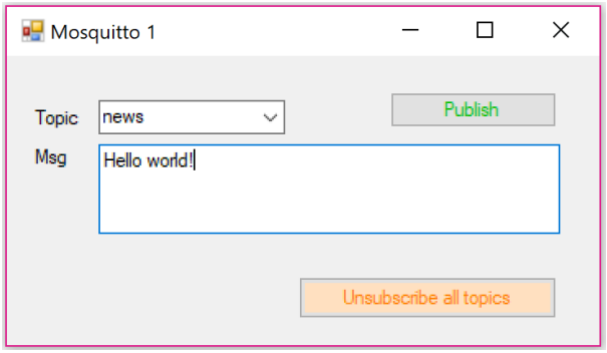

Figure 1 – Mosquitto 1 interface.

4.2. Example 2 – Message broker in another computer

These examples use console applications to exchange messages using a "far way" message broker (e.g.: 192.168.237.155).

Note: Instead of using a local instance you may also user an online server as e.g.: test.mosquitto.org

Application A – listens to events from news and complaints topics

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Net;
using System.Text;
using System.Threading.Tasks;
using uPLibrary.Networking.M2Mqtt;
```

```
using uPLibrary.Networking.M2Mqtt.Messages;

namespace MosquittoA
{
    class Program
    {

        static void client_MqttMsgPublishReceived(object sender, MqttMsgPublishEventArgs e)
        {
            Console.WriteLine("Received = " + Encoding.UTF8.GetString(e.Message) +
                              " on topic " + e.Topic);
        }

        static void Main(string[] args)
        {
            MqttClient mClient = new MqttClient(IPAddress.Parse("192.168.237.155"));
            string[] mStrTopicsInfo = { "news", "complaints" };

            mClient.Connect(Guid.NewGuid().ToString());
            if (!mClient.IsConnected)
            {
                Console.WriteLine("Error connecting to message broker...");
                return;
            }

            //Specify events we are interest on
            //New Msg Arrived
            mClient.MqttMsgPublishReceived += client_MqttMsgPublishReceived;

            //Subscribe to topics
            byte[] qosLevels = { MqttMsgBase.QOS_LEVEL_EXACTLY_ONCE,
                                 MqttMsgBase.QOS_LEVEL_EXACTLY_ONCE};  //QoS – depends on the topics number
            mClient.Subscribe(mStrTopicsInfo, qosLevels);

            Console.ReadKey();

            if (mClient.IsConnected)
            {
                mClient.Unsubscribe(mStrTopicsInfo); //Put this in a button to see notif!
                mClient.Disconnect(); //Free process and process's resources
            }

        }
    }
}
```

Application B – Posts messages to news topic

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Net;
using System.Text;
using System.Threading.Tasks;
using uPLibrary.Networking.M2Mqtt;

namespace MosquittoB
{
    class Program
    {
        static void Main(string[] args)
        {
            MqttClient mcClient = new MqttClient(IPAddress.Parse("192.168.237.155"));
            string[] mStrTopicsInfo = { "news", "complaints" };

            mcClient.Connect(Guid.NewGuid().ToString());
            if (!mcClient.IsConnected)
            {
                Console.WriteLine("Error connecting to message broker...");
                return;
            }
```

```
            mcClient.Publish("news", Encoding.UTF8.GetBytes("Hello World!"));

            Console.ReadKey();

            if (mcClient.IsConnected)
            {
                mcClient.Unsubscribe(mStrTopicsInfo); //Put this in a button to see notify!
                mcClient.Disconnect(); //Free process and process's resources
            }
        }
    }
}
```

4.2.1. Exercise

Implement the same behaviour but using Windows forms application for the Subscriber (MosquittoA) and the Publisher (MosquittoB) application. Test these applications running two or more subscribers instance and at least on publisher instance. Use a "far way" message broker domain (e.g. 192.168.237.155).

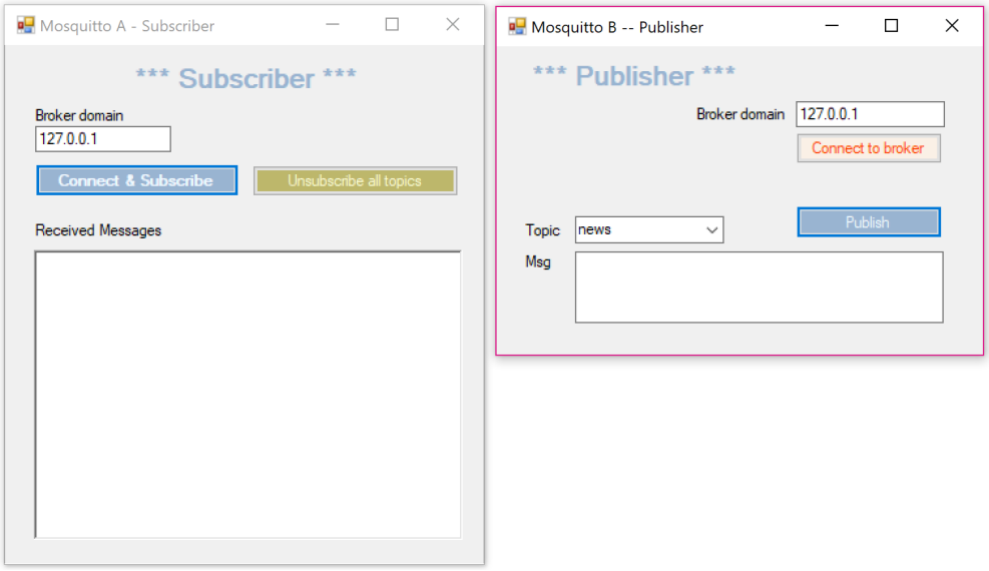

Figure 2 - Subscriber (MosquittoA) and Publisher (MosquittoB) applications

4.3. Example 3 – chat application

This example shows a simple chat application (MosquittoChatClient) where each user can specify its nickname, its classroom, and its avatar. Here is a snapshot of the application GUI:

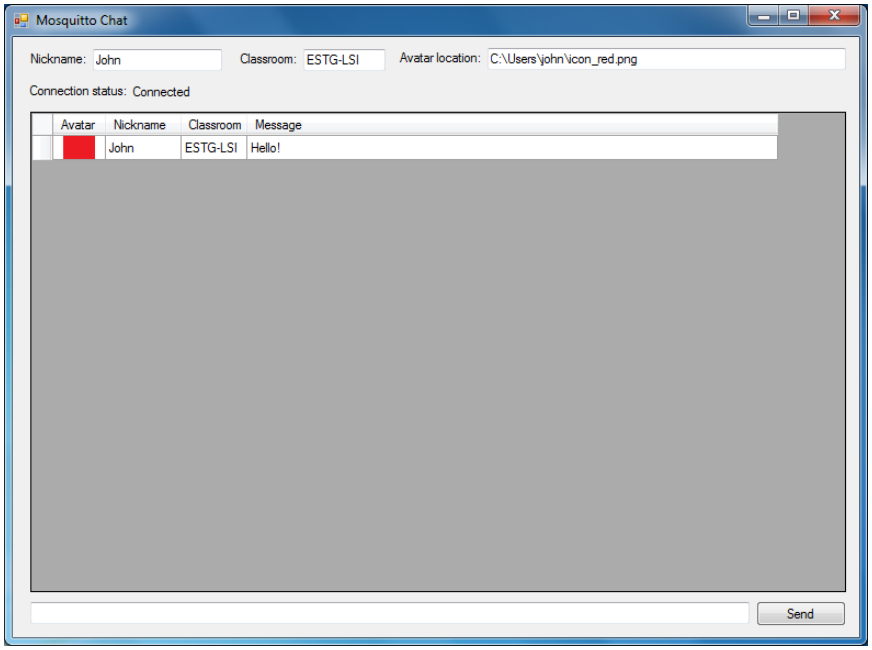

Fig. 3 – MosquittoChatClient's GUI

Download MosquittoChatClient from moodle and give it a try with your fellows.

4.4. Exercise 1

a) The MosquittoChatClient app seems to work well, however it has a (big) weakness… Which kind of weakness is it?

b) Fix the application main weakness.

c) Implement a Web Service that returns the channel name to the MosquittoChatClient Application if the specified user is known. It is not mandatory to implement a system to register users (this can be made by hand). The user login and password are stored in an XML file.

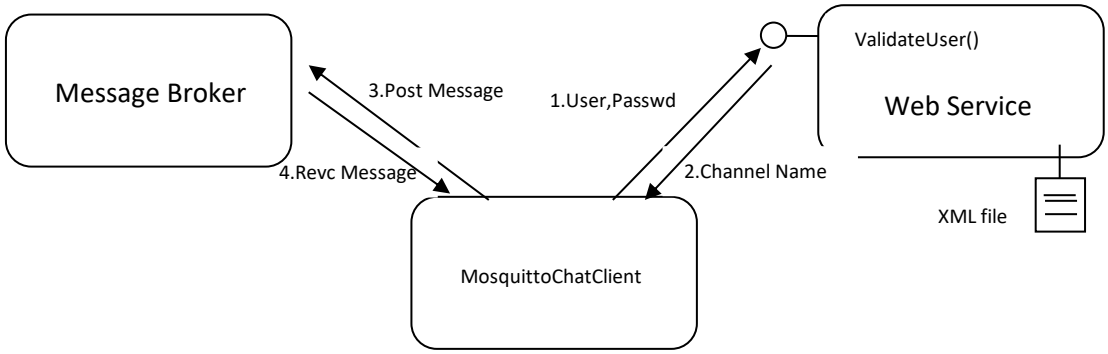

Fig. 4 – High-level overview of the new MosquittoChatClient's architecture