

# IoT Application Protocols

## RestFulAPI, MQTT, CoAP, XMPP

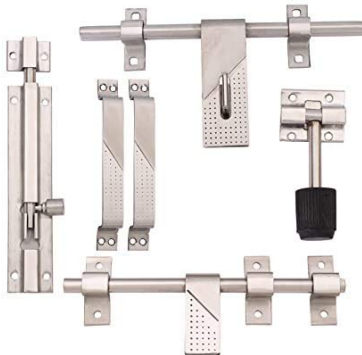
Dr. Munesh Singh

Indian Institute of Information Technology Design and Manufacturing,  
Kancheepuram, Chennai, Tamil-Nadu 600127

March 18, 2021



- Application programming Interface
- We build application and user interact with application using interface
- Interfaces are kind of door to an application
- Example: **Interface: Representational State Transfer API**



# Representational State Transfer

Resource	Verb	Result
/pizzas	GET	<i>All</i> the pizzas
/pizzas/:name	GET	<b>One</b> special pizza
/pizzas	POST	<b>Make</b> a pizza
/pizzas/:name	PUT	<b>Change</b> the pizza
/pizzas/:name	DELETE	<b>Delete</b> the pizza

To sum it up!:

GET	→	READ
POST	→	CREATE
PUT	→	UPDATE
DELETE	→	DELETE



# Representational State Transfer

- It is an architectural style that defines a set of rules in order to create Web Services.
- **Principles of REST API**
  - **Stateless** URL is used to uniquely identify the resource and the body holds the state of the requesting resource
  - **Client-Server** architecture enables a uniform interface and separates clients from the servers.
  - **Uniform Interface** To obtain the uniformity throughout the application
    - Resource identification
    - Resource Manipulation using representations
    - Self-descriptive messages
    - Hypermedia as the engine of application state
  - **Cacheable** saving a new request to the server



# Rest APIs (GET Method)

```
from flask import Flask, jsonify, request
app = Flask(__name__)

languages=[{'name':'javascript'},{'name':'python'},{'name':'ruby'}]

@app.route('/', methods=['GET'])
def returnall():
    return jsonify({'languages':languages})

if __name__ == '__main__':
    app.run(host='0.0.0.0',port=8080)

@app.route('/lang/<string:name>', methods=['GET'])
def returnone(name):
    langs =[language for language in languages if language['name']==name]
    return jsonify({'language':langs[0]})
```



# Rest APIs (POST Method)

```
@app.route('/lang', methods=['POST'])
def addone():
    language = {'name': request.json['name']}
    languages.append(language)
    return jsonify({'languages': languages})
```

POST 127.0.0.1:8080/lang No Environment

127.0.0.1:8080/lang Save

POST 127.0.0.1:8080/lang Send

Params Authorization Headers (8) **Body** Pre-request Script Tests Settings Cookies

none form-data x-www-form-urlencoded raw binary GraphQL JSON Beautify

1 {"name": "C++"}

Body Cookies Headers (4) Test Results Status: 200 OK Time: 6 ms Size: 235 B Save Response

Pretty Raw Preview Visualize JSON

```
10  {
11    "name": "ruby"
12  },
13  {
14    "name": "C++"
15  }
15 ]
```



# Rest APIs (PUT Method)

```
@app.route('/lang/<string:name>', methods=['PUT'])
def editone(name):
    langs = [language for language in languages if language['name']==name]
    langs[0]['name']=request.json['name']
    return jsonify({'language':langs[0]})
```

PUT 127.0.0.1:8080/lang/... Overview + ... No Environment

127.0.0.1:8080/lang/javascript Save

PUT 127.0.0.1:8080/lang/javascript Send

Params Authorization Headers (8) **Body** Pre-request Script Tests Settings Cookies

● none ● form-data ● x-www-form-urlencoded ● raw ● binary ● GraphQL JSON Beautify

```
1 { "name": "C" }
```

Body Cookies Headers (4) Test Results 200 OK 9 ms 174 B Save Response

Pretty Raw Preview Visualize JSON

```
1 {
2   "language": {
3     "name": "C"
4   }
5 }
```



# Rest APIs (DELETE Method)

```
@app.route('/lang/<string:name>', methods=['DELETE'])
def removeone(name):
    langs = [language for language in languages if language['name'] == name]
    languages.remove(langs[0])
    return jsonify({'languages': languages})
```

DEL 127.0.0.1:8080/lang... Overview DEL 127.0.0.1:8080/lang... + ... No Environment

127.0.0.1:8080/lang/javascript Save

DELETE 127.0.0.1:8080/lang/python Send

Params Authorization Headers (8) Body ● Pre-request Script Tests Settings Cookies

Query Params

KEY	VALUE	DESCRIPTION	...	Bulk Edit
Key	Value	Description		

Body Cookies Headers (4) Test Results 200 OK 6 ms 180 B Save Response

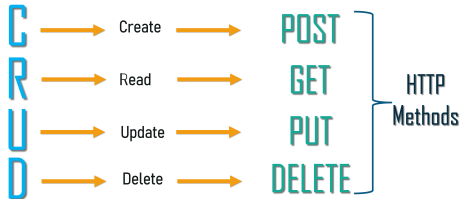
Pretty Raw Preview Visualize JSON

```
1
2  "languages": [
3    {
4      "name": "ruby"
5    }
6  ]
```





# Methods of REST API



URI

`http://weather.example.com/oaxaca`

Identifies

Resource

*Oaxaca Weather Report*

Represents

Representation

Metadata:  
Content-type:  
`application/xhtml+xml`

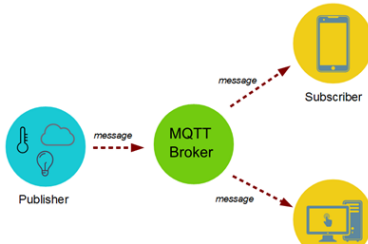
Data:

```
<!DOCTYPE html PUBLIC "...  
  "http://www.w3.org/...  
<html xmlns="http://www...  
<head>  
<title>5 Day Forecaste for  
Oaxaca</title>  
...  
</html>
```

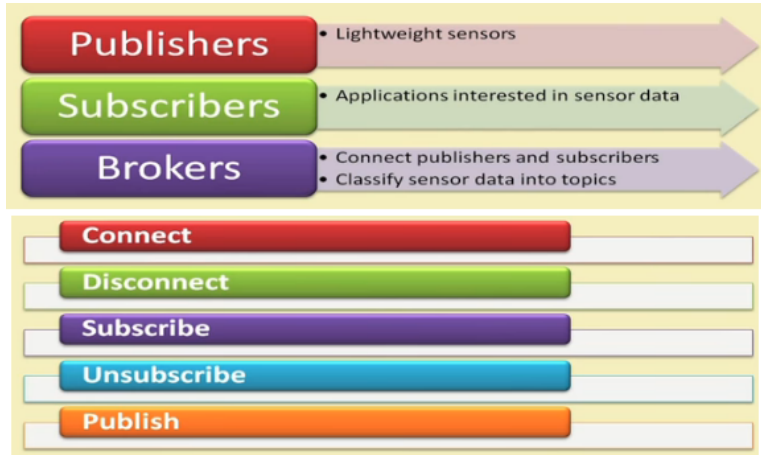


# Message Queuing Telemetry Transport

- It is a lightweight protocol built on the top of TCP/IP.
- Lightweight: In software/programs, lightweight specify the characteristic of low memory and CPU usage.
- MQTT allows simple and efficient data transmission from sensors to IoT (Internet of Things) networks.
- It is an asynchronous messaging scheme
- Based on publisher-subscriber (pub-sub) paradigm
- It is an open standard (OASIS)
- It evolved from message queuing frameworks
- In general, implemented using Client-server paradigm!



# MQTT Component & Methods



# Services Offered by MQTT Broker

## Services

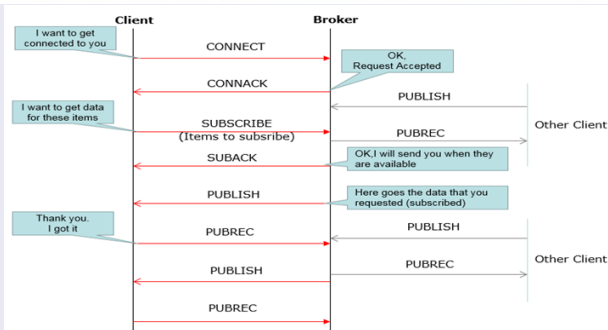
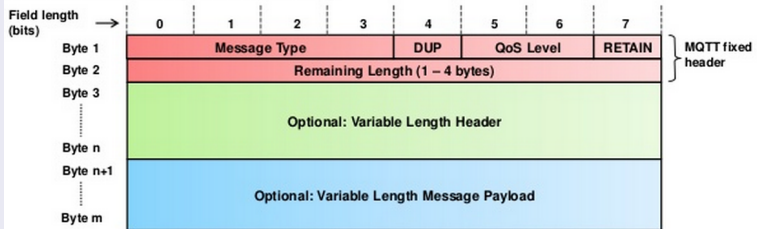
- Message Store and Forward
- Message Filtering
- Message Prioritization
- Messaging with different QoS

## MQTT Message Transport

- MQTT rides on TCP
- Broker uses TCP port number 1883 for normal connection
- Broker uses TCP port number 8883 for secure connection
- Secure communication uses TLS



# MQTT Formate



# Configure CloudMQTT Broker

The screenshot displays the CloudMQTT console interface. At the top, the browser address bar shows the URL `api.cloudmqtt.com/console/82644103/details`. The page header includes the CloudMQTT logo, a user profile dropdown for 'Munesh Singh', and a list of open browser tabs. The left sidebar contains a navigation menu with options: DETAILS (selected), SETTINGS, CERTIFICATES, USERS & ACL, BRIDGES, AMAZON KINESIS STREAM, WEBSOCKET UI, STATISTICS, CONNECTIONS, and LOG. The main content area is titled 'Details' and features an 'Instance info' section with the following configuration:

Field	Value
Server	<code>tailor.cloudmqtt.com</code>
User	<code>jhopphcz</code> <span>Restart</span>
Password	<code>gWsgC6...</code> <span>Copy</span>
Port	<code>17534</code>
SSL Port	<code>27534</code>
Websockets Port (TLS only)	<code>37534</code>
Connection limit	<code>5</code>

To the right of the instance info is an 'Active Plan' section featuring a 'Cute Cat' avatar and an 'Upgrade Instance' button. A red chat bubble icon is located in the bottom right corner of the console area.



# CoAP- Constrained Application Protocol.

- Web transfer protocol for use with constrained nodes and networks.
- **Designed for Machine to Machine (M2M)** applications such as smart energy and building automation
- Based on **Request-Response Model** between end-point
- Client-Server interaction is asynchronous over a datagram oriented transport protocol such as UDP
- CoAP is designed to enable low-power sensors to use RESTful services while meeting their power constraints.

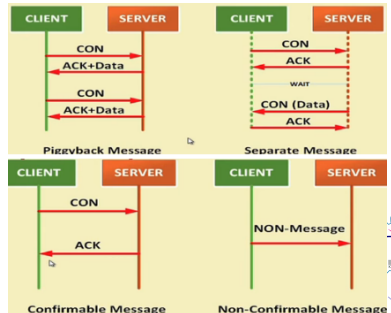
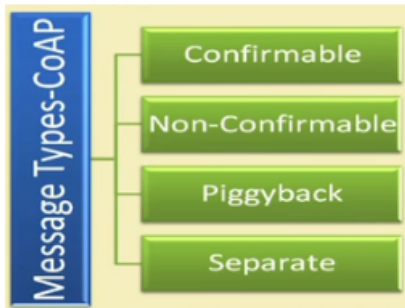
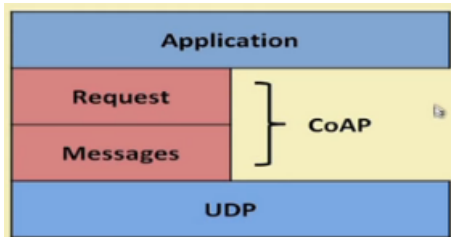


- Built over UDP, instead of TCP (which is commonly used with HTTP) and has a light mechanism to provide reliability.
- CoAP architecture is divided into two main sub-layers:
  - Messaging
  - Request/response
- The messaging sub-layer is responsible for reliability and duplication of messages, while the request/response sub-layer is responsible for communication
- CoAP has four messaging modes:
  - Confirmable
  - Non-confirmable
  - Piggyback
  - Separate





# CoAP Position & CoAP Message Types



# CoAP Request-Response Model

- Confirmable and non-confirmable modes represent the reliable and unreliable transmissions, respectively, while the other modes are used for request/response.
- Piggyback is used for client/server direct communication where the server sends its response directly after receiving the message, i.e., within the acknowledgment message.
- On the other hand, the separate mode is used when the server response comes in a message separate from the acknowledgment, and may take some time to sent by the server.
- Similar to HTTP, CoAP utilizes GET, PUT, PUSH, DELETE messages requests to retrieve, create, update, and delete, respectively

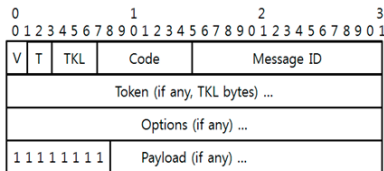


# COAP Features

- COAP is a light weight Web protocol developed using RESTful architecture
- A stateless derivative of HTTP with intermediaries and caching
- Request-Response model with support for Asynchronous response
- Optional ACK for both request and response Support for block transfer for both request and response
- COAP operates on resources;
- Aids M2M communication with Resource Discovery
- Optimized for Constrained (power, lossy) networks
- Reduced overheads and parsing complexity.
- URL and content-type support.
- Support for the discovery of resources provided by known CoAP services.
- Simple subscription for a resource, and resulting push notifications.



# CoAP Formate



**coap-URI** = "coap:" "://" host [ ":" port ] path-abempty [ "?" query ]

**coaps-URI** = "coaps:" "://" host [ ":" port ] path-abempty [ "?" query ]

CoAP message header	Description
Ver	It is 2 bit unsigned integer. It mentions CoAP version number. Set to one.
T	It is 2 bit unsigned integer. Indicates message type viz. confirmable (0), non-confirmable (1), ACK (2) or RESET(3).
TKL	It is 4 bit unsigned integer, Indicates length of token (0 to 8 bytes).
Code	It is 8 bit unsigned integer, It is split into two parts viz. 3 bit class (MSBs) and 5 bit detail (LSBs).
Message ID	16 bit unsigned integer. Used for matching responses. Used to detect message duplication.

Methods: GET, PUT, DELETE, and POST (the first 3 are idempotent)



- To install Twisted and txThings on your Raspberry Pi shell, run the following commands:
  - **sudo pip install twisted**
  - **sudo pip install txthings**
- **Install on the PC**
  - 1 `git clone git://github.com/siskin/txThings.git`
  - 2 `pip install twisted`
  - 3 `pip install txThings`
- **Install wireshark:**
  - `sudo apt-get install wireshark -y`
  - `sudo setcap 'CAP_NET_RAW+eip CAP_NET_ADMIN+eip' /usr/bin/dumpcap`

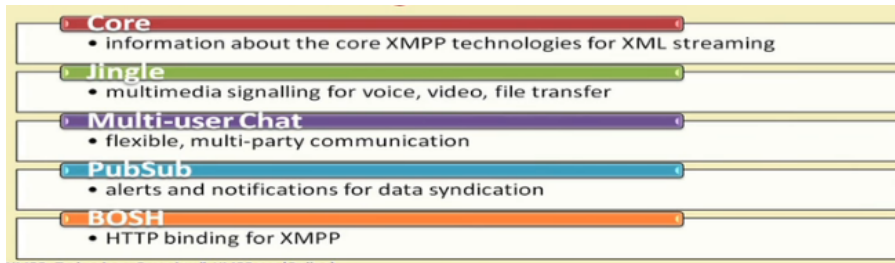


# XMPP-Extensible Messaging and Presence Protocol XMPP

- A communication protocol for message-oriented middle-ware based on XML (Extensible Markup Language).
- Real-time exchange of structured data.
- It is an open standard protocol
- XMPP uses a client-server architecture.
- As the model is decentralized, no central server is required.
- XMPP provides for the discovery of services residing locally or across a network.
- Open means to support machine-to-machine or peer-to-peer communications across a diverse set of networks.
- Security-Authentication, encryption, etc.
- Flexibility-Supports interoperability



# Core XMPP Technologies



## Weaknesses

- Does not support QoS.
- Text based communications induces higher network overheads.
- Binary data must be first encoded to base64 before transmission.



# Component of XMPP protocol

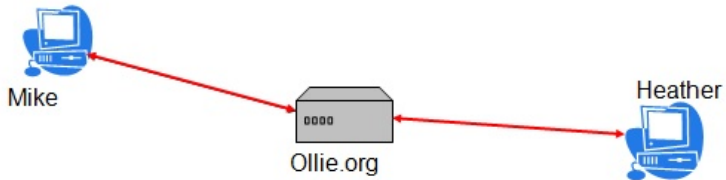
- **Extensible:** It can be customized to individual user needs.
- **Messaging:** It uses short messages as method of communication between client (i.e. user) and server.
- **Presence:** It is reactive to presence of the user and user status.
- **Protocol:** It is not a language. It is open platform which is constantly evolving. It is based on XML and it is asynchronous.





# XMPP Core Overview

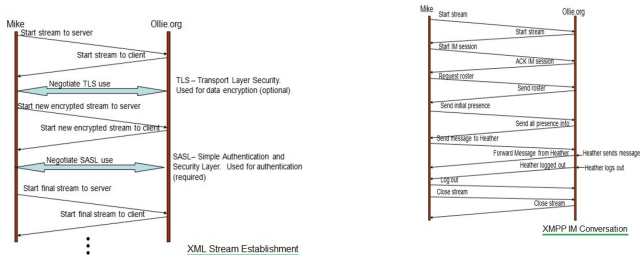
- XMPP is an open protocol for streaming XML elements in order to exchange messages and presence information in close to real time.
- XMPP protocol works as per typical client server architecture
- XMPP client utilizes XMPP server using TCP socket.



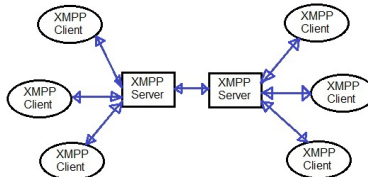
XMPP and TCP



# XMPP Protocol between XMPP Server and XMPP Client



- Instant messaging is used as means for immediate message transmission and reception to online users.



- Following are features of XMPP protocol used between XMPP client and XMPP server for communication:
  - XMPP uses port number 5222 for client to server (C2S) communication.
  - XMPP uses port number 5269 for server to server (S2S) communication.
  - Discovery and XML streams are used for S2S and C2S communications.
  - Uses security mechanisms such as TLS and SASL.
  - There are no intermediate servers for federation unlike E-mail.
- A sample implementation on XMPP using unofficial official IoT XEPs(XMPP Extensions)
- Today we will be using XEP0030(service discovery), XEP0323(IoT sensor data) and XEP325(IoT control) for this sample.



## ● STEP 1: First we need to setup a XMPP server.

- Setting up XMPP Openfire Server and expose via RESTful APIs
- <http://www.igniterealtime.org/projects/openfire/>
- Now setup wizard should be appeared on the web browser.  
Select "English" as the language and click "continue". On the second screen enter the following;
  - Domain – > 127.0.0.1 #or enter the IP assigned on ifconfig.
  - Admin Console Port – > 9090 #http login
  - Secure Admin Console Port – > 9091 #https login
  - Property Encryption via – > Blowfish

Server Settings

Below are host settings for this server. Note: the suggested value for the domain is based on the network settings of this machine.

Domain: 127.0.0.1 ⓘ

Admin Console Port: 9090 ⓘ

Secure Admin Console Port: 9091 ⓘ

Property Encryption via: ⓘ

☒ Blowfish

☐ AES

Property Encryption Key:  ⓘ

Continue



## ● Step 2:

- On the next step select "Embedded Database" and click "Continue". This will simplify the database connection process and make use of its HSQLDB internally embedded DB.
- On the production environment you must select "Standard Database Connection" and configure a separate database.



**Database Settings**

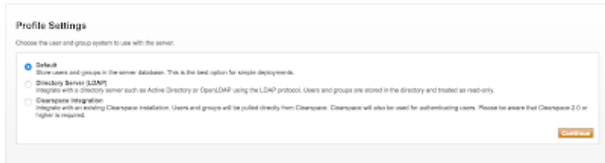
Choose how you would like to connect to the Openfire database.

☐ Standard Database Connection  
Use an external database with the built-in connection pool.

☒ Embedded Database  
Use an embedded database, powered by HSQLDB. This option requires no external database configuration and is an easy way to get up and running quickly. However, it does not offer the same level of performance as an external database.

[Continue](#)

- For the profile settings; select "Default". this option will keep the user information on the connected database.



**Profile Settings**

Choose the user and group system to use with the server.

☒ Default  
Store users and groups in the server database. This is the best option for simple deployments.

☐ Directory Server (LDAP)  
Integrate with a directory server such as Active Directory or OpenLDAP using the LDAP protocol. Users and groups are stored in the directory and treated as read-only.

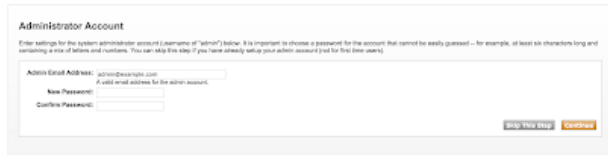
☐ Cleanspace Integration  
Integrate with an existing Cleanspace installation. Users and groups will be pulled directly from Cleanspace. Cleanspace will also be used for authenticating users. Please be aware that Cleanspace 2.0 or higher is required.

[Continue](#)



### • Step 3:

- On the next step; enter an "email" and a valid password for the admin account. Click "Continue".



**Administrator Account**

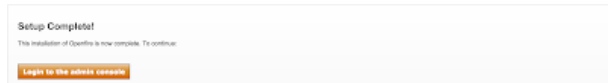
Enter settings for the system administrator account (username of "admin") below. It is important to choose a password for the account that cannot be easily guessed -- for example, at least six characters long and containing a mix of letters and numbers. You can skip this step if you have already setup your admin account (not for first time users).

Admin Email Address:   
A valid email address for the admin account.

New Password:   
Confirm Password:

[Skip This Step](#) [Continue](#)

- Once you are done, It will display "Setup Complete!" message. Then click on "Log to the admin console".

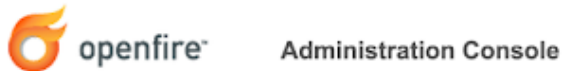


**Setup Complete!**

This installation of Openfire is now complete. To continue:

[Login to the admin console](#)

- Once you are redirected, enter the username as "admin" and the password will be the same you gave on the previous



The image shows the login form for the Openfire Administration Console. It has a "username" field with "admin" entered, a "password" field, and a "Login" button.



## ● Creating a New User

- Navigate into "Users/Groups" tab. Select "Create New User" on the left menu pane.

Use the form below to create a new user.

**Create New User**

Username \*

Admin

Name

Admin

Email

Password \*

admin

Confirm Password \*

admin

Is Administrator?

☐ (Grants admin access to Openfire)

Create User

Create & Create Another

Cancel

\* Required fields

**User Summary**

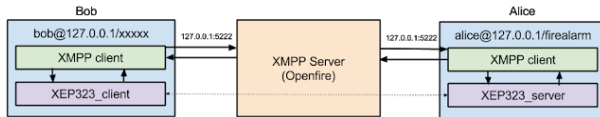
Total Users: 2 - Sorted by Username - Users per page: 100

	Online	Username	Name	Groups	Created	Last Logout	Edit	Delete
1		admin	Administrator	None	Aug 13, 2015			
2		asaka	Asaka Puro	None	Aug 13, 2015			

- Deploying REST-API plugin to manage users
  - When integrating with other servers, manually creating each user on the openfire is not scalable or feasible approach.
  - Openfire plugins <http://www.igniterealtime.org/projects/openfire/plugins.jsp>
  - REST-API plugin provides functionalities to manage users through HTTP calls <http://www.igniterealtime.org/projects/openfire/plugins/restapi/readme.html>



- RestApi plugin deployment
  - In order to install plugin, you can just copy "restApi.jar" file into \$OPEN-FIRE-HOME/plugins folder.
  - " /usr/local/openfire/plugins"
  - To configure REST API, Go to Server -> Server Settings. Then REST API from the left menu pane.
- Once you have setup Openfire server you need to create two user accounts "bob" and "alice".
  - Our user story is that "alice" has a sensor device called "firealarm" and "bob" needs to get sensor information (e.g. temperature) of the "firealarm"



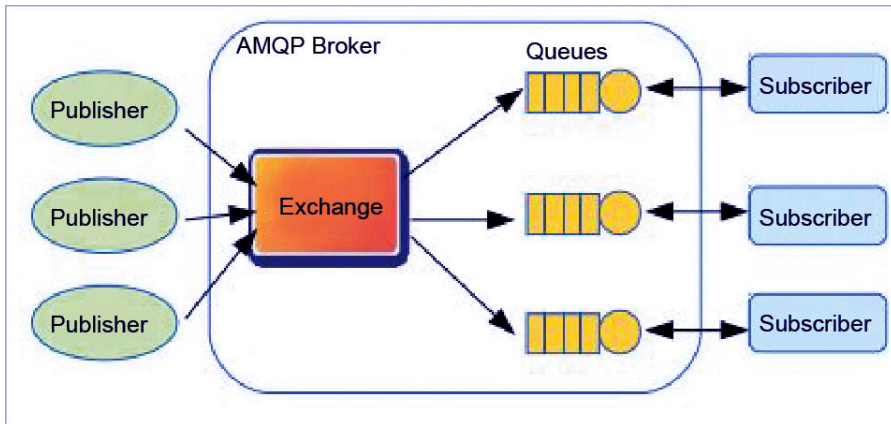


- Copy paste below python scripts into two files "xmpp\_server.py" and "xmpp\_client.py".
  - [https://drive.google.com/file/d/1-cI0sbNKYBV4eWorckDj\\_Vf\\_ejSmfmbn/view?usp=sharing](https://drive.google.com/file/d/1-cI0sbNKYBV4eWorckDj_Vf_ejSmfmbn/view?usp=sharing)
  - We need to install SleekXMPP library
  - Enter following to install SleekXMPP from PyPI **pip install sleekxmpp**
  - Once Openfire server is up and running you can issue following commands;
    - 1 **python xmpp\_server.py -j "alice@127.0.0.1" -p "password" -n "firealarm"**
    - 2 on other terminal
    - 3 **python xmpp\_client.py -j "bob@127.0.0.1" -p "password" -c "alice@127.0.0.1/firealarm" -q**
  - On Ubuntu Linux Server, you can quickly install the OpenFire server
    - `wget https://www.igniterealtime.org/downloadServlet?filename=openfire/openfire_4.6.0_all.deb -O openfire_4.6.0_all.deb`
    - `sudo apt install ./openfire_4.6.0_all.deb`
    - `systemctl status openfire`
    - `http://ip-address-of-your-server:9090`

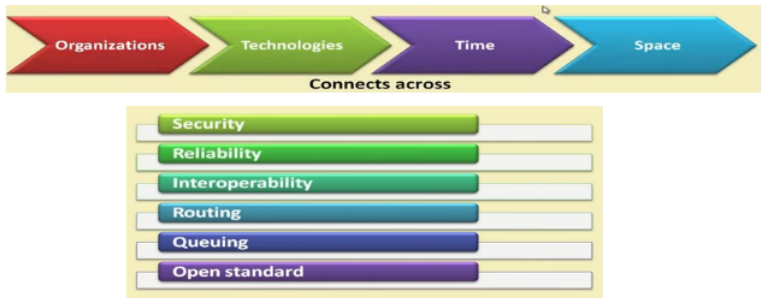


- **Advanced Message Queuing Protocol.**
- **Open standard for passing business messages** between applications or organizations.
- Connects between systems and business processes
- It is a binary application layer protocol
- Basic unit of data is a frame
- ISO standard: ISO/IEC 19464





# AMQP Features



# Message Delivery Guarantees

- At-most-once
  - each message is delivered once or never
- At-least-once
  - each message is certain to be delivered, but may do so multiple times.
- Exactly-once
  - message will always certainly arrive and do so only once

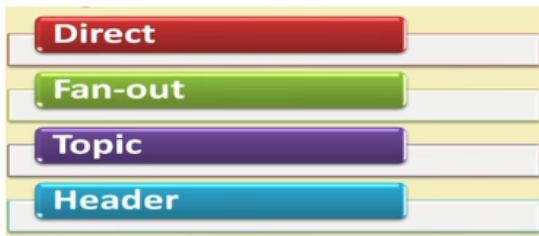
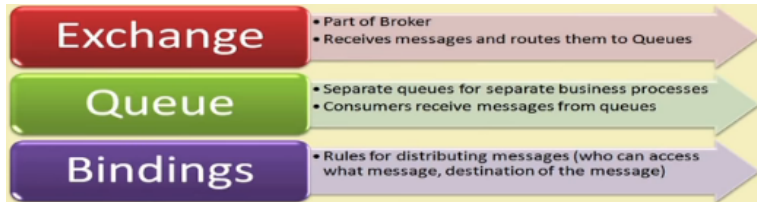


# AMQP Frame Types

- Nine AMQP frame types are defined that are used to initiate, control and tear down the transfer of messages between two peers:
  - Open (connection open)
  - Begin (session open)
  - Attach (initiate new link)
  - Transfer (for sending actual messages)
  - Flow (controls message flow rate)
  - Disposition (informs the changes in state of transfer)
  - Detach (terminate the link)
  - End (session close)
  - Close (connection close)



# Components & AMQP Exchanges



# AMQP Features

- Targeted QoS (selectively offering QoS to links)
- Persistence (Message delivery guarantees)
- Delivery of messages to multiple consumers
- Possibility of ensuring multiple consumption
- Possibility of preventing multiple consumption
- High speed protocol





# Applications

- Monitoring and global update sharing.
- Connecting different systems and processes to talk to each other.
- Allowing servers to respond to immediate requests quickly and delegate time consuming tasks for later processing.
- Distributing a message to multiple recipients for consumption.
- Enabling offline clients to fetch data at a later time.
- Introducing fully asynchronous functionality for system
- Increasing reliability and uptime of application deployments.



# carrot - AMQP Messaging Framework for Python

- carrot is an AMQP messaging queue framework.
- AMQP is the Advanced Message Queuing Protocol, an open standard protocol for message orientation, queuing, routing, reliability and security.
- carrot has pluggable messaging back-ends, so it is possible to support several messaging systems.
  - AMQP (py-amqplib, pika), STOMP (python-stomp)
  - In-memory backend for testing purposes, using the Python queue module  
<https://docs.python.org/3/library/queue.html>
  - Several AMQP message broker implementations exists
    - RabbitMQ,
    - ZeroMQ and Apache ActiveMQ
    - Youll need to have one of these installed, personally weve been using RabbitMQ.



# Installation of carrot

- You can install carrot either via the Python Package Index (PyPI) or from source.
  - `pip install carrot`
  - `easy_install carrot`
  - `$ python setup.py build`  
`# python setup.py install # as root`
- **There are some concepts you should be familiar with before starting:**
  - **Publishers:** Publishers sends messages to an exchange.
  - **Exchanges:** Exchanges are named and can be configured to use one of several routing algorithms
    - The exchange routes the messages to consumers by matching the routing key in the message with the routing key the consumer provides when binding to the exchange.
  - **Consumers** Consumers declares a queue, binds it to a exchange and receives messages from it.



- **Queues**

- Queues receive messages sent to exchanges.
- The queues are declared by consumers

- **Routing keys**

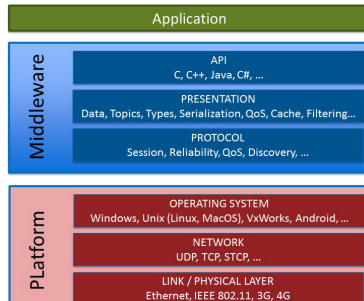
- Every message has a routing key.
- The interpretation of the routing key depends on the exchange type.
- There are four default exchange types defined by the AMQP standard, and vendors can define custom types (so see your vendors manual for details).
  - **Direct exchange** Matches if the routing key property of the message and the routing\_key attribute of the consumer are identical.
  - **Fan-out exchange** Always matches, even if the binding does not have a routing key.
  - **Topic exchange** Matches the routing key property of the message by a primitive pattern matching scheme.

- <https://pythonhosted.org/carrot/introduction.html>

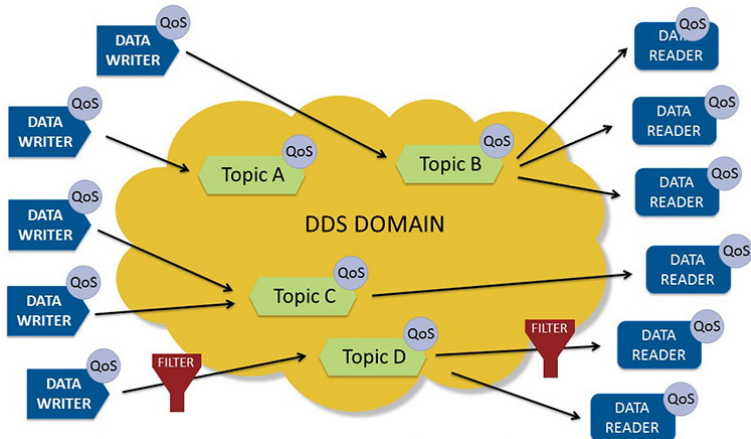


# Data Distribution Service

- The Data Distribution Service (DDS) for real-time systems is an Object Management Group (OMG) machine-to-machine
- It implements a publishsubscribe pattern for sending and receiving data, events, and commands among the nodes.
- It integrates the components of a system together, providing low-latency data connectivity, extreme reliability, and a scalable architecture that business and mission-critical Internet of Things (IoT) applications need.

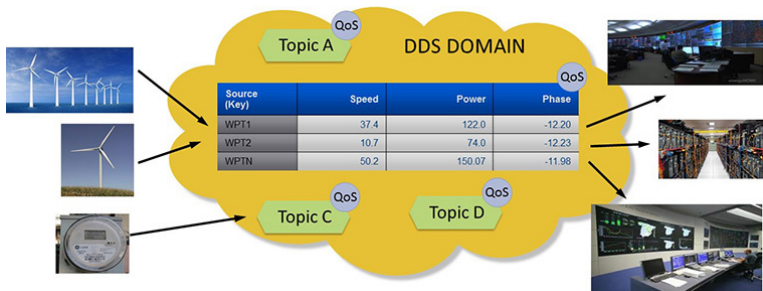


- DDS provides QoS-controlled data-sharing. Applications communicate by publishing and subscribing to Topics identified by their Topic name.
- Data Centric
  - DDS is uniquely data centric, which is ideal for the Industrial Internet of Things.



# Global Data Space

- DDS conceptually sees a local store of data called the global data space.
- To the application, the global data space looks like local memory accessed via an API.
- In reality, DDS sends messages to update the appropriate stores on remote nodes.

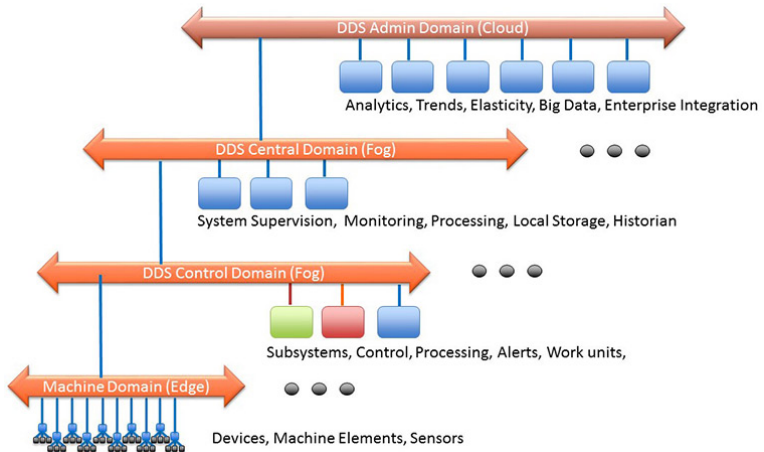


- The data can also be shared with flexible Quality of Service (QoS) specifications including reliability, system health (liveliness), and even security.
- **Dynamic Discovery**
  - DDS provides Dynamic Discovery of publishers and subscribers.
  - Dynamic Discovery makes your DDS applications extensible.
  - This means the application does not have to know or configure the endpoints for communications because they are automatically discovered by DDS.
  - This can be completed at runtime and not necessarily at design or compile time, enabling real plug-and-play for DDS applications.
  - This dynamic discovery goes even further than discovering endpoints





# Scalable Architecture



- **Security**

- Protecting mission-critical Industrial IoT environments requires security that scales from edge to cloud, across systems and suppliers.
- DDS includes security mechanisms that provide authentication, access control, confidentiality, and integrity to the information distribution.
- DDS Security uses a decentralized peer-to-peer architecture that provides security without sacrificing real-time performance.

- **Implementation**

- <https://github.com/atolab/pydds>
- <https://www.rti.com/blog/introducing-rti-labs-and-connector-for-connext-dds-with>



*Thank You*

