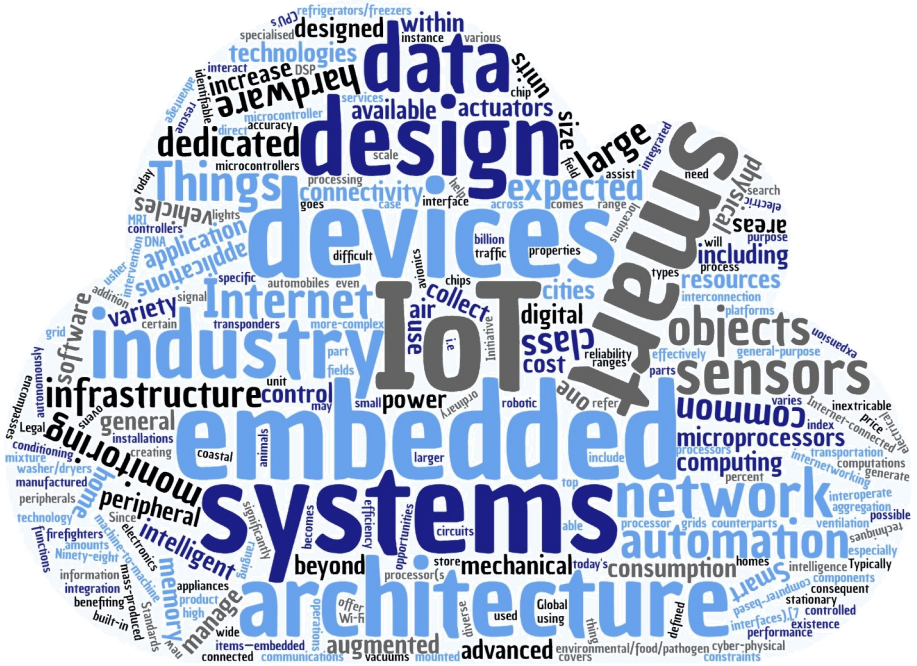
Associate Professorship of Embedded Systems and Internet of Things
Department of Electrical and Computer Engineering
Technical University of Munich

Lecture
IoT Remote Lab

05 – Protocols

Ege Korkan



Zoom Guidelines

- The lectures and tutor sessions happen on Zoom meetings following the link sent to you via email.
- Participatition to Zoom sessions is optional
- You can choose a random string for your name
- The Zoom chat will not be recorded and we will not save the chat.
- All the participants except the lecturer is muted. The participants are free to unmute. You can also go to participants, click the hand icon to raise your hand.
- You can also do other things, like asking me to go slower. I have a separate window where I look at the requests from the participants.
- You can ask quick questions in Zoom chat or use the Tweedback link provided in each session.

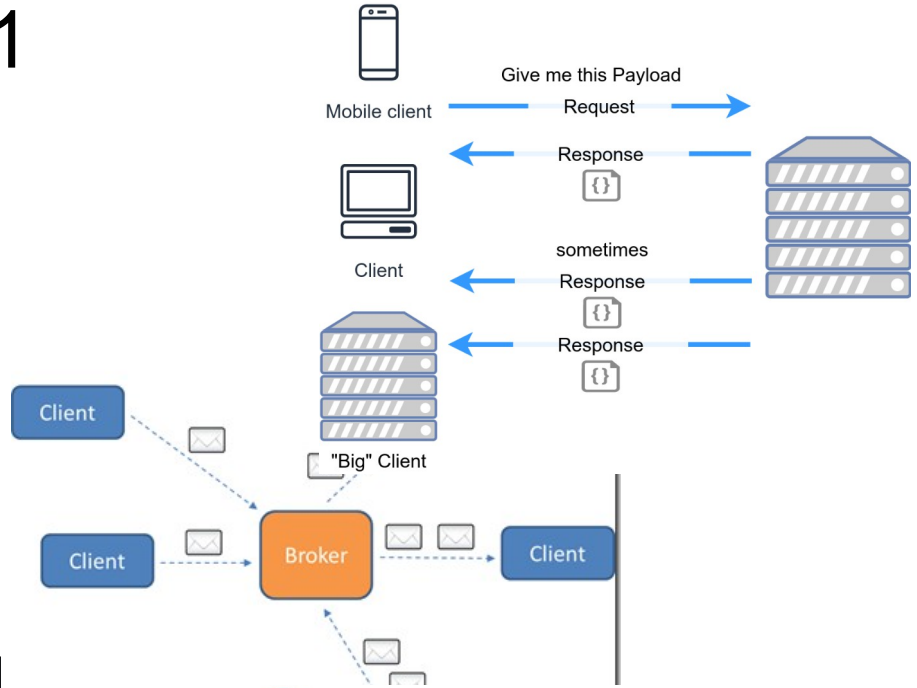
Tweedback for Real-time Q&A During the Lecture

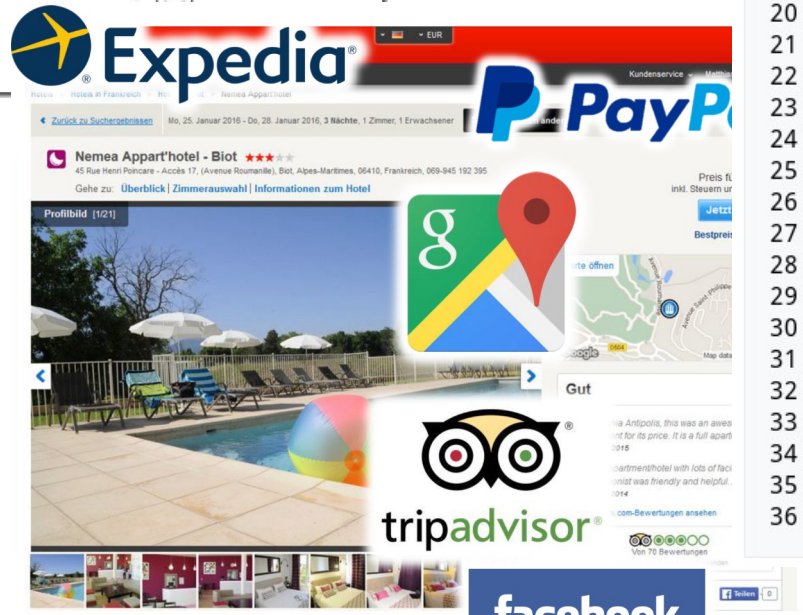
https://tweedback.de/pcv1/chatwall

Recap: Season 1

```
{
  "productId": 1,
  "productName": "An ice sculpture",
  "price": 12.50,
  "tags": [ "cold", "ice" ],
  "dimensions": {
    "length": 7.0,
    "width": 12.0,
    "height": 9.5
  },
  "warehouseLocation": {
    "latitude": -78.75,
    "longitude": 20.4
  }
}
```

```
{
  "$schema": "http://json-schema.org/draf
  "$id": "http://example.com/product.sche
  "title": "Product",
  "description": "A product from Acme's c
  "type": "object",
  "properties": {
    "productId": {
      "description": "The unique identifi
      "type": "integer"
    },
    "productName": {
      "description": "Name of the product
      "type": "string"
    },
    "price": {
      "description": "The price of the pr
      "type": "number",
      "exclusiveMinimum": 0
    },
```






```
{
    "@context": "https://www.w3.org/2019/wot/td/v1",
    "id": "urn:dev:ops:32473-WoTLamp-1234",
    "title": "MyLampThing",
    "securityDefinitions": {
        "basic_sc": {"scheme": "basic", "in":"header"}
    },
    "security": ["basic_sc"],
    "properties": {
        "status" : {
            "type": "string",
            "forms": [{
                "href": "https://mylamp.example.com/status",
                "htv:methodName":"GET"
            }]
        }
    },
    "actions": {
        "toggle" : {
            "forms": [{
                "href": "https://mylamp.example.com/toggle",
                "htv:methodName":"POST"
            }]
        }
    },
    "events":{
        "overheating":{
            "data": {"type": "string"},
            "forms": [{
                "href": "https://mylamp.example.com/oh",
                "htv:methodName":"GET",
                "subprotocol": "longpoll"
            }]
        }
    }
}
```

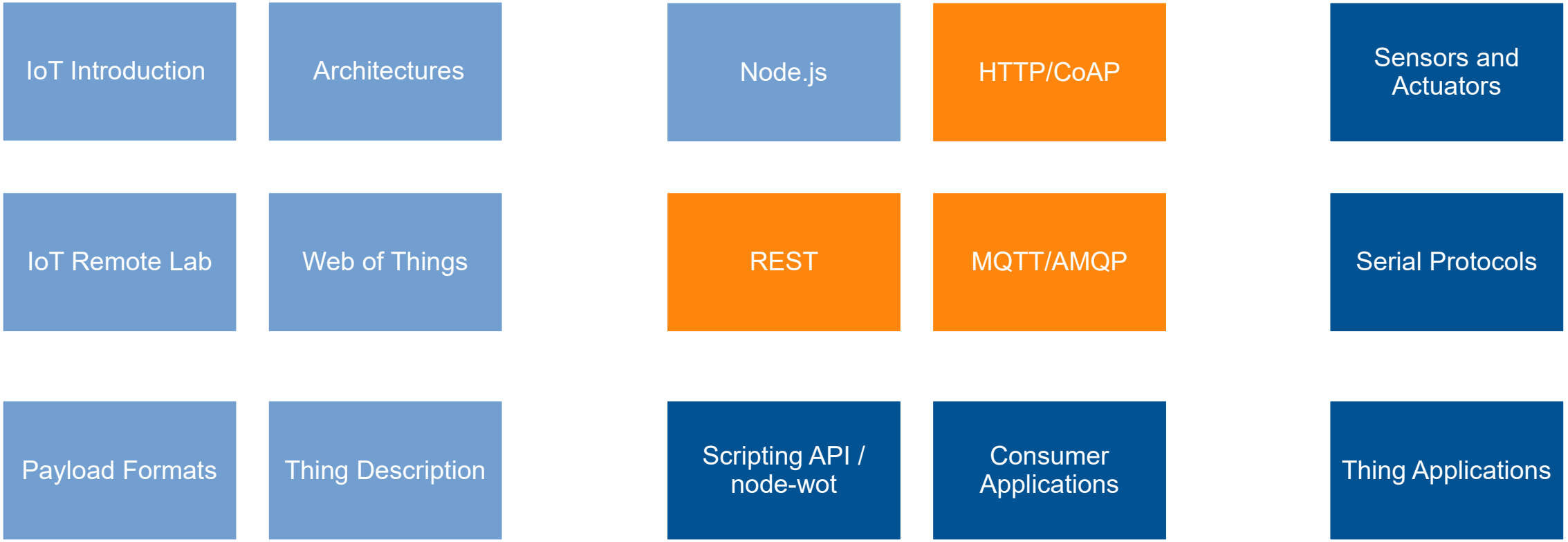
Recap of Last Lecture

```
async function myFunction() {

  let promise = new Promise((resolve, reject) => {
    setTimeout(() => resolve("done!"), 1000)
  });

  let result = await promise; // wait until the promise resolves (*)

  console.log(result); // "done!"
}

myFunction();
```

Recap of Last Lecture

```
1 class Student {
2     fullName: string;
3     constructor(public firstName: string, public middleInitial: string, public lastName:
  string {
4         this.fullName = firstName + " " + middleInitial + " " + lastName;
5     }
6 }
7
8 interface Person {
9     firstName: string;
10    lastName: string;
11 }
12
13 function greeter(person: Person) {
14    return "Hello, " + person.firstName + " " + person.lastName;
15 }
16
17 let user = new Student("John", "M.", "Doe");
18
19 console.log(greeter(user));
20
```

Course Contents



Deliverable 1

- Deliverable 1 is created on Artemis.
- You should sign up yourself.
- Deadline of Deliverable 1 is **16.12.2020 23:59 (11:59 pm)**.
- Late submissions will not be accepted at all, you simply lose 30% of your grade.
- Tests are not running yet.

Deliverable 2

- Deliverable 2 is about interacting with devices
- It will be available on Artemis as well.
- Deadline of Deliverable 2 is **27.01.2021 23:59 (11:59 pm)**.
- Late submissions will not be accepted at all.

Protocols

We will be looking at :

- HTTP
- CoaP
- MQTT
- AMQP

There are of course more protocols that are on the same level but these should provide a good understanding that can be applied to the other ones

HTTP

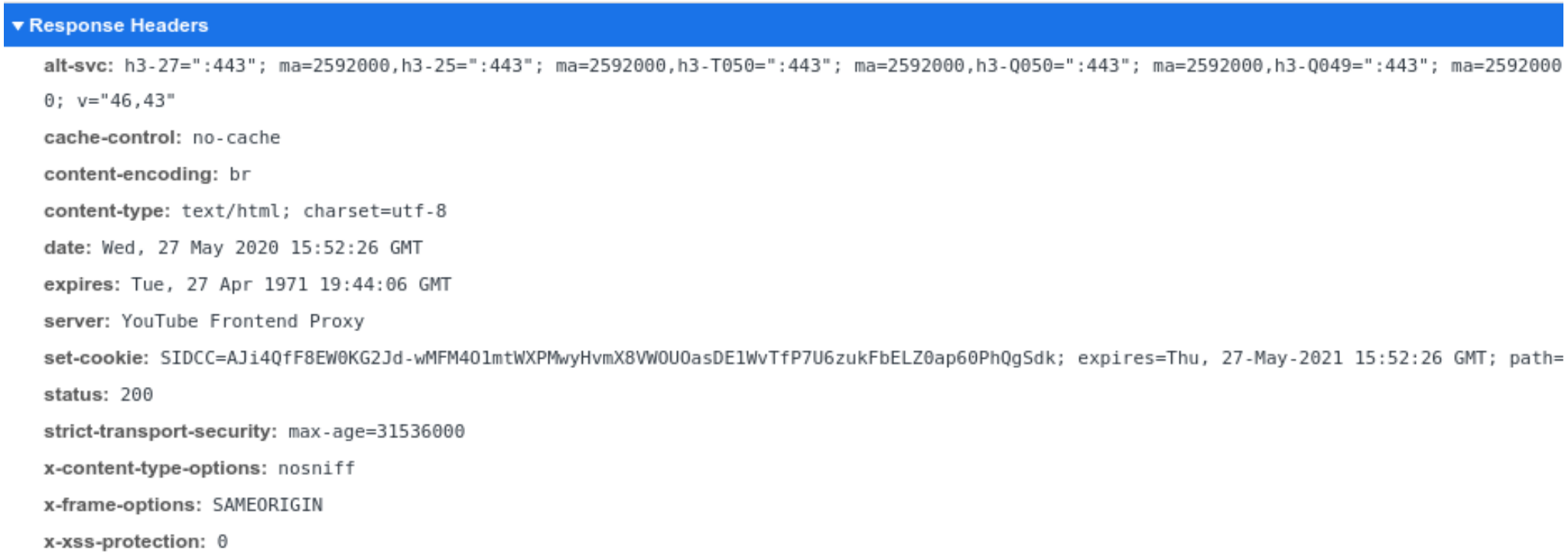
- Everybo
- Questior is a valid URL in it
  - Your

▼ Request Headers

```
:authority: www.youtube.com
:method: GET
:path: /
:scheme: https
accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,image/apng,*/*;q=0.8,application/signed-exchan
accept-encoding: gzip, deflate, br
accept-language: en-US,en;q=0.9,fr-FR;q=0.8,fr;q=0.7,de;q=0.6,tr;q=0.5
cache-control: max-age=0
cookie: VISITOR_INFO1_LIVE=ZwwmbH9_3Vc; amplitude_id_7acb8df1d57962e98163686ab410fa23_crystal_ce_amplitudeyoutube.com=e
TUzMzE1Njk2MTM5MSwibGFzdEV2ZW50VGltZSI6MTUzMzE1Njk2MTM5MSwiZXZlbnRJZCI6MCwiaWRlbnRpZnlJZCI6MCwic2VxdWVuY2VOdW1iZXIiOjB9
xrQ/AEDW4F46ED4WQohjG; SAPISID=VWwiAWeKyhqxdz_R/ACX3iGip-zb34amOC; __Secure-HSID=A6Sjn-3uSkXoL3sLe; __Secure-SSID=AgbP
N_INFO=AFmmF2swRgIhAL65oUOM_HLfdLhvL3SsaQyuN758ZxHrqMZ53aax-xkoAiEA_-5L78meiNz3Uzc0d_F1AJvW6ABcxp_PNGoyhPjf1wA:QUQ3MjNm
bV92ZTFScjBKcUJhRlRIdGphUWdzU2dWdWFwTWNrM1JTTXNjVTFSQkxOekhaalNzeW5mMDJucUVoYUlOMjY0YnNLNEJMWjJNRlZKOWNHUnpobFlZ; __Se
gkoDc2g2wxeDBZOebCcd39JYbPKcbSz5g.; YSC=ZuT99_oLRTo; PREF=cvdm=grid&al=en+fr+tr&f1=50000000&ldlayout=PLxPCxMDxISXOCxIC
dnt: 1
sec-fetch-dest: document
sec-fetch-mode: navigate
sec-fetch-site: none
sec-fetch-user: ?1
upgrade-insecure-requests: 1
user-agent: Mozilla/5.0 (X11; Linux x86_64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/81.0.4044.138 Safari/537.36
x-client-data: CJG2yQEIpLbJAQipncoBCNCvygEIvLDKAQiatcoBCO21ygEIjrrKAQjqu8oBGLy6ygEYmr7KAQ==
```

HTTP

- Then gets a response



HTTP

- What about when you fill a form (like when registering to a Website)?
  - Either a GET request is sent with form data in the URL or a POST/PUT request with data in the body

    More here:
    https://developer.mozilla.org/en-US/docs/Learn/Forms/Sending_and_retrieving_form_data

HTTP

What we have seen:

- There are lots of header options in requests and responses
  - Data for the application logic should go in the body (payload) but this is not necessarily the case
  - https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers
- There are different methods. Saying an HTTP request is not enough
  - These different methods have also different meanings → Protocol semantics
  - https://developer.mozilla.org/en-US/docs/Web/HTTP/Methods
- There are different status codes in the responses
  - These also have different meanings! → Protocol semantics
  - https://developer.mozilla.org/en-US/docs/Web/HTTP/Status

HTTP

Any idea how we can do eventing in a **browser environment**?

Possible solutions:

- Websockets: https://developer.mozilla.org/en-US/docs/Web/API/WebSocket
- Longpoll: https://www.ably.io/topic/long-polling
- Servent Sent Events (event source): https://html.spec.whatwg.org/multipage/server-sent-events.html

HTTP

We do not need to master HTTP :)

(I don‘t)

HTTP

Takeaways:

- Learn your methods, minimum is GET, PUT, POST
- Think of what should be in the body and what can be in the headers
- Eventing:
  - Not high speed (like a button press) → Long poll
  - Unidirectional from the Server → SSE (or WS)
  - Bi-directional and not REST → WS

HTTP in WoT

Vocabulary term	*Description*	*Assignment*	*Type*
`htv:methodName`	HTTP method name (Literal).	optional	`string` (one of `"GET"`, `"PUT"`, `"POST"`, `"DELETE"`, `"PATCH"`)
`htv:headers`	HTTP headers sent with the message.	optional	array of `htv:MessageHeader`
`htv:fieldName`	Header name (Literal), e.g., `"Accept"`, `"Transfer-Encoding"`.	mandatory within `htv:MessageHeader`	`string`
`htv:fieldValue`	Header value (Literal).	mandatory within `htv:MessageHeader`	`string`

HTTP in WoT

op value	*Default Binding*
readproperty	"htv:methodName": "GET"
writeproperty	"htv:methodName": "PUT"
invokeaction	"htv:methodName": "POST"
readallproperties	"htv:methodName": "GET"
writeallproperties	"htv:methodName": "PUT"
readmultipleproperties	"htv:methodName": "GET"
writemultipleproperties	"htv:methodName": "PUT"

REST

- For the remainder, we will do a more question/answer session.
- You will see examples that contain HTTP but these principles can be mapped to other protocols.
- My oral explanations are important, not everything is in the slides but I will attach a tutorial from our SADES course.

REST

- What is REST?
  - It is an architecture, not a protocol, not an API.
  - Even though it is used a lot in Web APIs, it is not necessarily only for Web since it is just an architecture style for network-based software architectures.
  - It imposes some constraints on the server-client architecture.
- It means ***Re****presentational* ***S****tate* ***T****ransfer* but this should not help you much
- Understanding TD will help with understanding REST and vice versa!

REST

- Do you know the principles of REST?
  - **Stateless:** Once you are using a server application, you cannot think that the server might remember you previous activities.
  - **Cacheable:** Server must indicate whether the response is cacheable.
  - **Layered System:** If a resource is accessed by many clients, it should be possible to add a middle layer that serves these multiple clients. E.g. The middle layer can cache the resource and send to every client who asks this resource, reducing the computing resource requirement of the initial resource server.
  - **Code-On-Demand:** Client can execute a code sent by the server to gain more functionality. This is an optional constraint.
  - **Uniform Interface:** The server provides the exact same interface to every client, no matter what the clients do with the resources. This principle gives the server-client decoupling.

REST

- Uniform interface has even its sub principles! Let‘s see:
  - **Identification of resources**
    - href in TD forms
  - **Manipulation of resources through these representations**
    - Interaction Affordances get modified through their forms but can be that an action changes a property value
  - **Self-descriptive messages**
    - `contentType` should be also present in messages in addition to the TD
  - **Hypermedia as the engine of application state (aka HATEOAS)**
    - Not possible in the TD at the moment but there are discussions on this! See more here: https://github.com/w3c/wot-thing-description/issues/899

REST

In a way, if all of REST‘s principles 100% were implemented by every API for any possible protocol, we would not need TDs that much.

(TD does not mean that your Thing implementation is RESTful though!)

Let‘s see how your API can be actually *made* RESTful and explore the **Richardson Maturity Model**

REST

- Level 0: Single URI, single method

  URI is http://example.com/myplatform and method is GET

```
{
  "readSwitch":{
    "switchId":2,
    "esi":"amazing",
    "reader":"gateway"
  }
}
```

```
{
  "toggleSwitch":{
    "switchId":2,
    "esi":"amazing",
    "reader":"gateway"
  }
}
```

REST

- Level 1: URI per resource, single method

  URI is http://example.com/myplatform/leftswitch and method is GET

```
{
  "readSwitch":{
    "esi":"amazing",
    "reader":"gateway"
  }
}
```

```
{
  "toggleSwitch":{
    "esi":"amazing",
    "reader":"gateway"
  }
}
```

REST

- Level 2: URI per resource, different methods for different operations

  URI is http://example.com/myplatform/leftswitch

GET

```
1 {
2   "esi":"amazing",
3   "reader":"gateway"
4 }
```

POST

```
1 {
2   "esi":"amazing",
3   "reader":"gateway"
4 }
```

REST

- Level 3: Discover the API like a human discovering a website

  A GET request to the API endpoint with URI http://example.com/myplatform

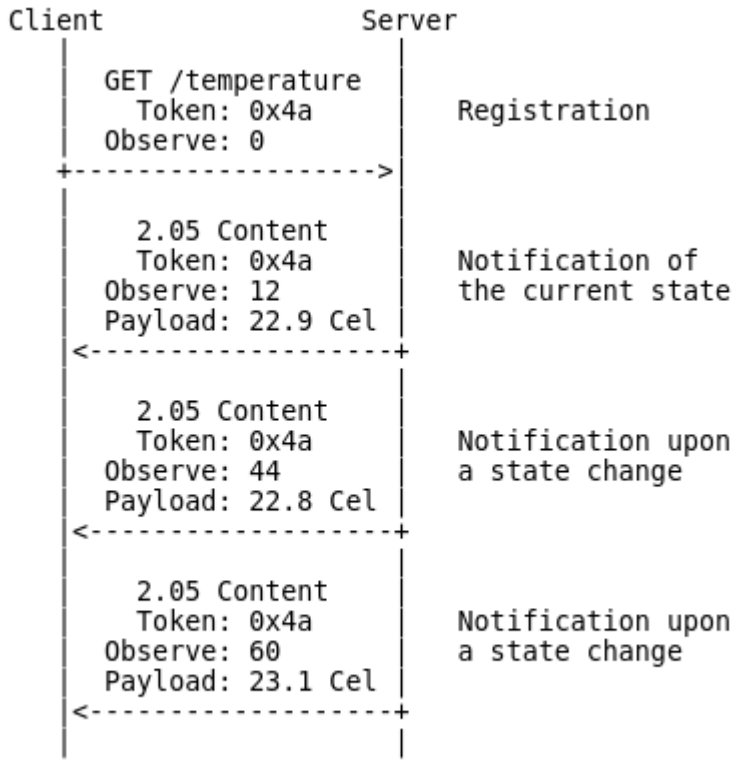
```
{
  "switches":[
    {
      "rel":"read",
      "href":"switch1"
    },
    {
      "rel":"read",
      "href":"switch2"
    },
    {
      "rel":"toggle",
      "href":"switch1"
    }
  ]
}
```

CoAP

- Server-client but for more constrained devices!
- Based on UDP instead of TCP that HTTP is based on
- Follows the RESTful design patterns so that it is easy to proxy from HTTP to CoAP and vice versa
- Similar Header options, status codes, method names
- Designed for smaller messages → Not loading websites
- Smaller header size, less header options → HTTP is 3 char (3 Byte) + Whitespace (1 Byte) vs 1 Byte Int
- Embedded observation mechanism

CoAP

CoAP Observe (https://tools.ietf.org/html/rfc7641)



CoAP in WoT (Vocabulary will be updated)

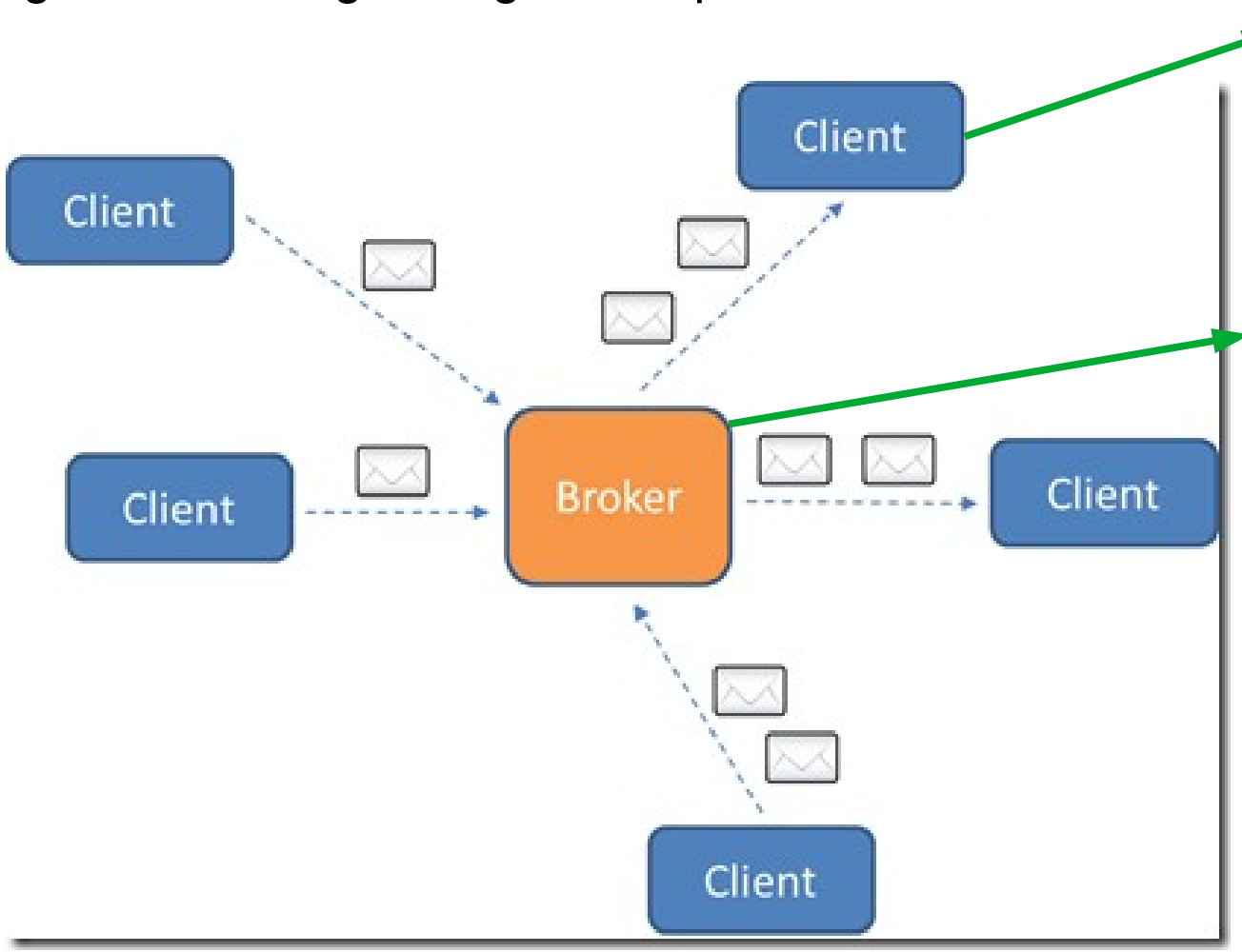
Vocabulary term	*Description*	*Assignment*	*Type*
`cov:methodName`	CoAP method name (Literal).	optional	`string` (one of `"GET"` (1), `"POST"` (2), `"PUT"` (3), `"DELETE"` (4), `"FETCH"` (5), `"PATCH"` (6), `"iPATCH"` (7))
`cov:options`	CoAP options sent with the message, e.g., `[ { "cov:optionName": "Accept", "cov:optionValue": 110 }]` to observe.	optional	array of `cov:MessageOption`
`cov:optionName`	Option name (Literal), see CoRE Parameters.	mandatory within `cov:MessageOption`	`string`
`cov:optionValue`	Header value (Literal).	mandatory within `cov:MessageOption`	`anyType`

CoAP in WoT (Vocabulary will be updated)

op value	*Default Binding*
readproperty	"cov:methodName": "GET"
writeproperty	"cov:methodName": "PUT"
observeproperty	"cov:methodName": "GET","subprotocol":"cov:observe"
unobserveproperty	"cov:methodName": "GET","subprotocol":"cov:observe"
invokeaction	"cov:methodName": "POST"
subscribeevent	"cov:methodName": "GET","subprotocol":"cov:observe"
unsubscribeevent	"cov:methodName": "GET","subprotocol":"cov:observe"
readallproperties	"cov:methodName": "GET"
writeallproperties	"cov:methodName": "PUT"
readmultipleproperties	"cov:methodName": "GET"
writemultipleproperties	"cov:methodName": "PUT"

MQTT

Starting with eventing from ground up!



Thing or Consumer

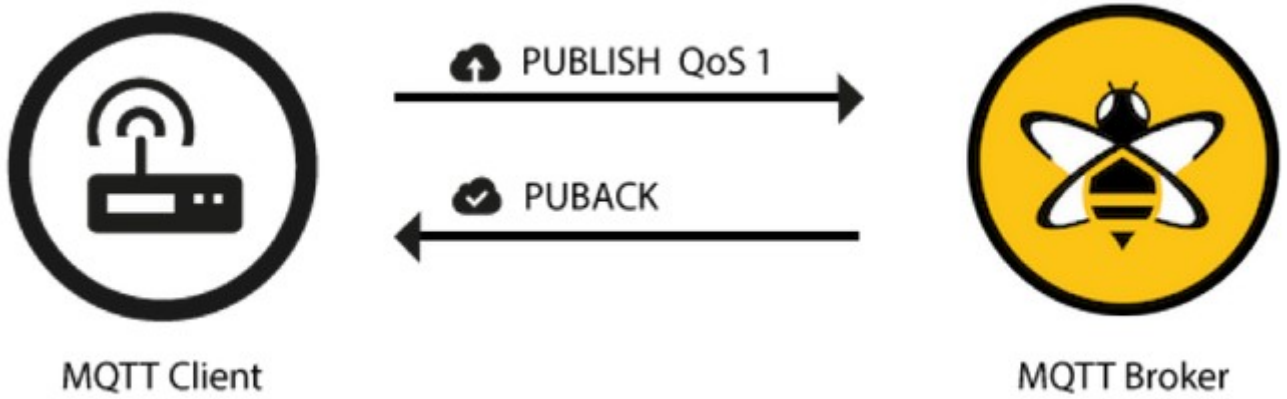
Not relevant for an application → Invisible in WoT

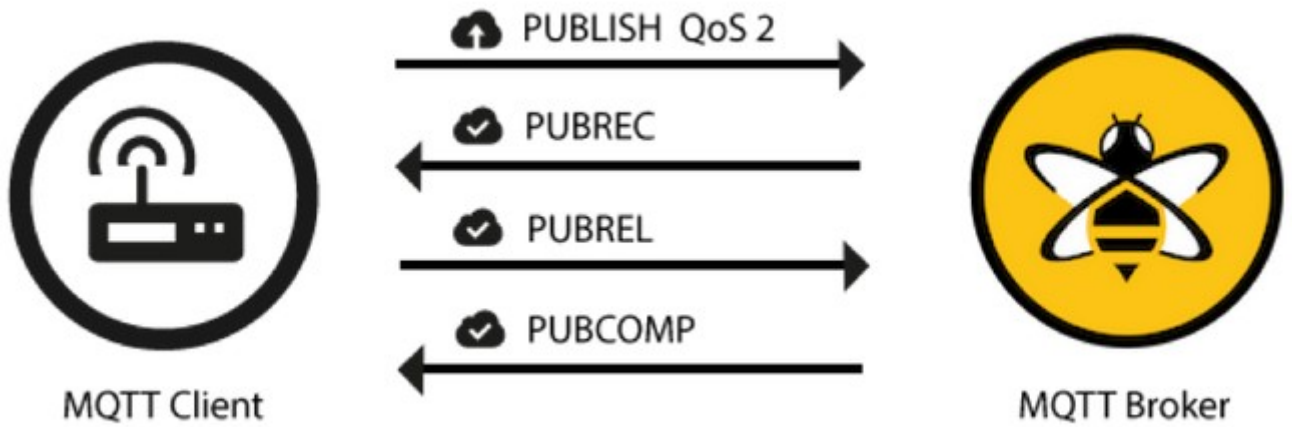
MQTT

- Standard at: https://docs.oasis-open.org/mqtt/mqtt/v5.0/mqtt-v5.0.html
- Publisher
  - QoS
  - Retain
  - Duplicate
  - More header options in the variable header

MQTT

- Layering is a bit weird! See more explanations at https://www.hivemq.com/blog/mqtt-essentials-part-6-mqtt-quality-of-service-levels/





MQTT in WoT

Vocabulary term	*Description*	*Assignment*	*Type*
`mqv:controlPacketValue`	MQTT Control Packet type (Literal).	optional	`string` (one of `"PUBLISH"` (3), `"SUBSCRIBE"` (8), `"UNSUBSCRIBE"` (10))
`mqv:options`	MQTT options sent with the message, e.g., `[ { "mqv:optionName": "qos", "mqv:optionValue": 1 }]`.	optional	array of `mqv:MessageOption`
`mqv:optionName`	Option name (Literal).	mandatory within `mqv:MessageOption`)	`string` (one of `"qos"`, `"retain"`, `"dup"`)
`mqv:optionValue`	Header value (Literal).	mandatory within `mqv:MessageOption`)	One of `0`, `1` or `2` (only for `qos`)

MQTT in WoT

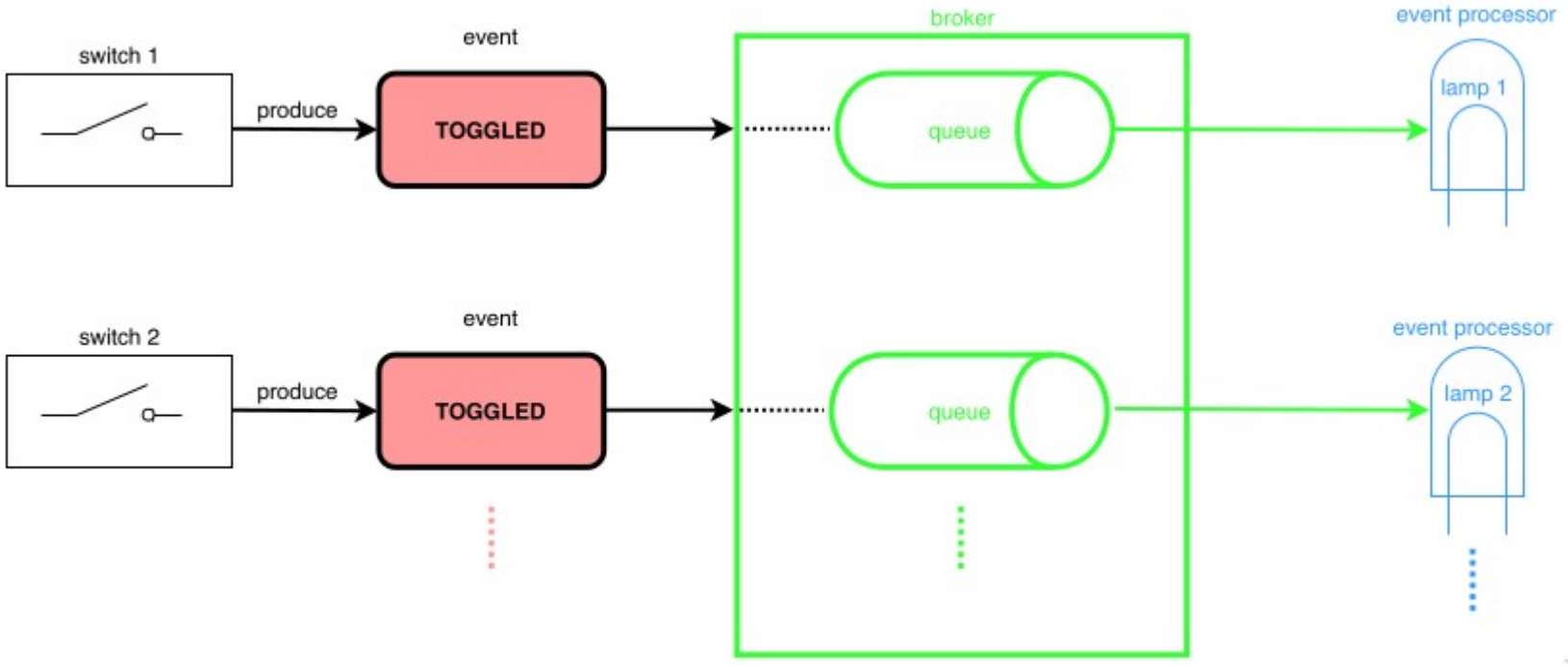
op value	*Default Binding*
readproperty	"mqv:controlPacketValue": "SUBSCRIBE",
writeproperty	"mqv:controlPacketValue": "PUBLISH"
observeproperty	"mqv:controlPacketValue": "SUBSCRIBE"
unobserveproperty	"mqv:controlPacketValue": "UNSUBSCRIBE"
invokeaction	"mqv:controlPacketValue": "PUBLISH"
subscribeevent	"mqv:controlPacketValue": "SUBSCRIBE"
unsubscribeevent	"mqv:controlPacketValue": "UNSUBSCRIBE"
readallproperties	"mqv:controlPacketValue": "SUBSCRIBE"
writeallproperties	"mqv:controlPacketValue": "PUBLISH"
readmultipleproperties	"mqv:controlPacketValue": "SUBSCRIBE"
writemultipleproperties	"mqv:controlPacketValue": "PUBLISH"

Protocols in WoT

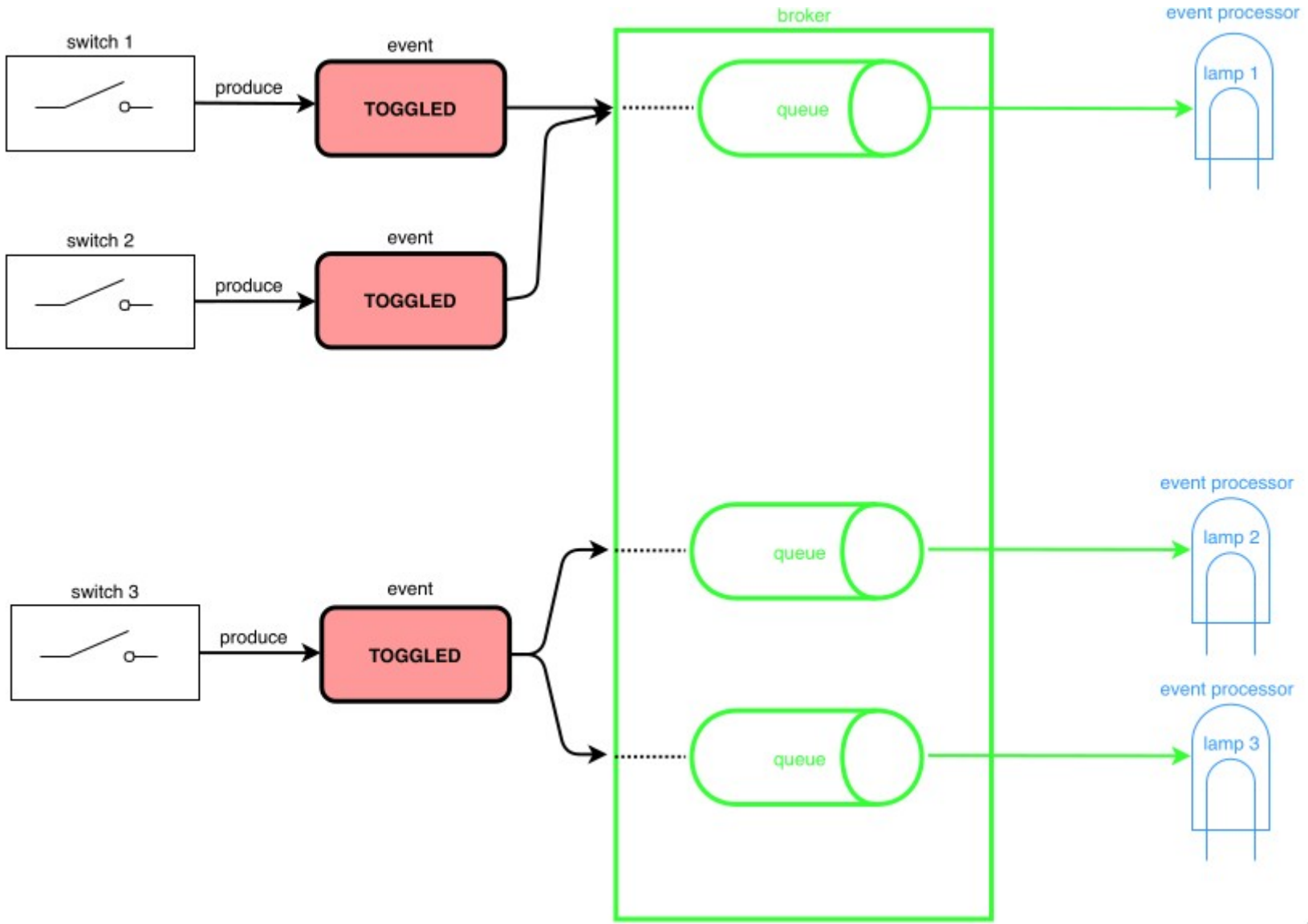
The screenshots you have seen are from the protocol bindings document, available at https://www.w3.org/TR/wot-binding-templates/

AMQP

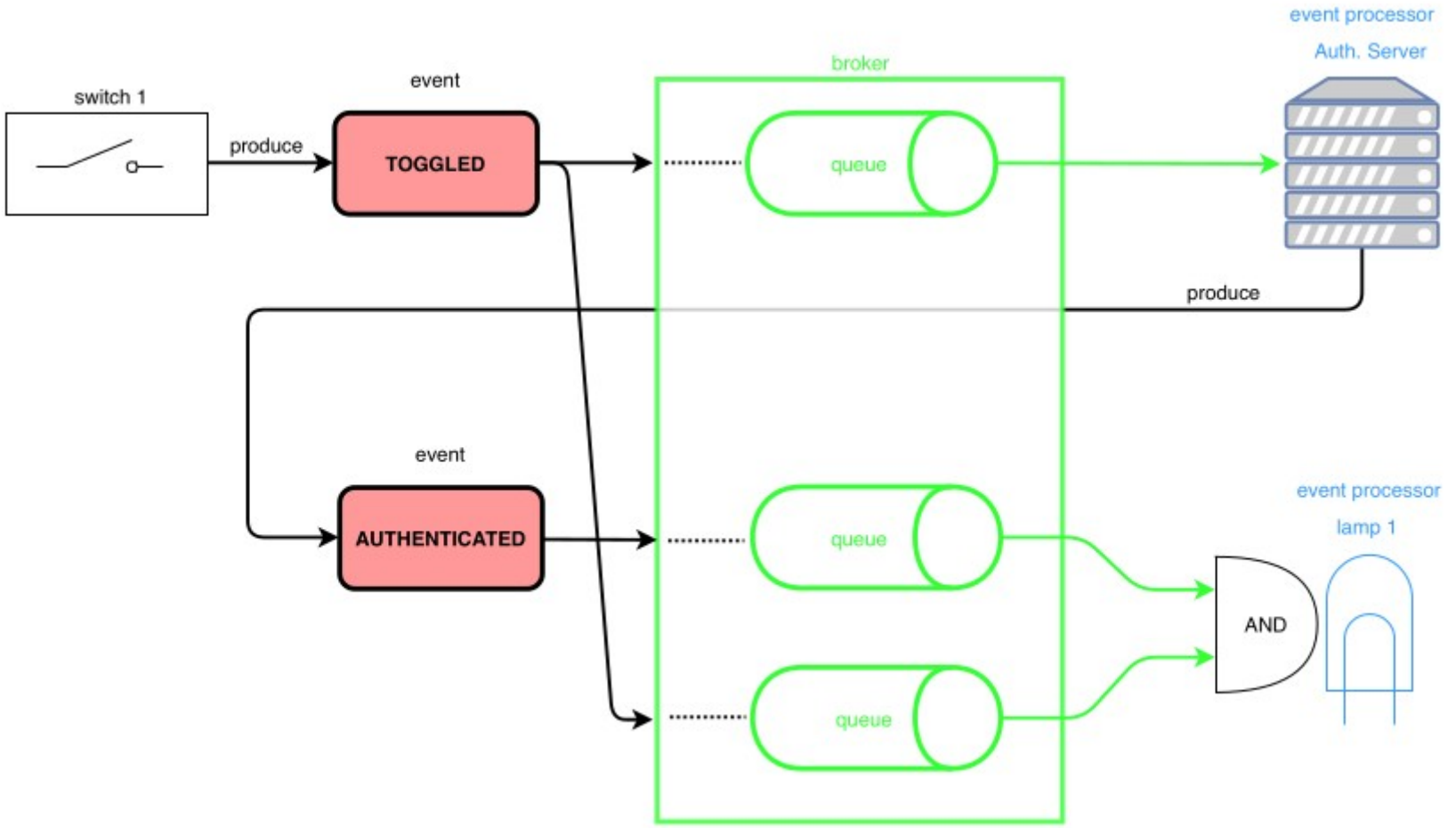
- In MQTT, the broker is very *simplistic,* i.e. it does nothing more than forwarding messages and sometimes storing them (retain)



AMQP


switch 1
produce
event
TOGGLED
switch 2
produce
event
TOGGLED
switch 3
produce
event
TOGGLED
broker
queue
queue
queue
event processor
lamp 1
event processor
lamp 2
event processor
lamp 3

AMQP


switch 1
produce
event
TOGGLED
broker
queue
event processor
Auth. Server
produce
event
AUTHENTICATED
queue
queue
event processor
lamp 1
AND

AMQP vs MQTT

- Notion of queues in AMQP and thus their management comes with more developer effort
- AMQP handles higher number of events in a more configurable way
- AMQP allows per topic authorization
- Not sure which one → Choose MQTT, Brokers can translate between the two as well!

Server-Client or Broker-based?

- Obvious architectural differences:
  - Broker based protocols are better for eventing where the Thing sends data without someone asking for it
  - Server-Client is better when there needs to be an agreement, i.e. invoking an action
  - Server and broker needs scaling, i.e. clients generally don‘t. However, normally brokers need less scaling since they do less work than a server
- Non-obvious differences:
  - Putting a server in a network requires a port open to the outside
  - A client to a broker looks like a client of a server in the network level → No need to keep a port open to the outside

Wrap-Up

- REST is an architectural principle that can be implemented with different protocols
- TD allows the use of different protocols by defining how they can be mapped to WoT operations
- Server-client or Broker-based protocols can be chosen based on your application requirements and they should be abstracted from the application logic