RIOT

... in the Internet of Things

Bachelor Project (PO)
Introduction to CoAP

Hamburg 27.03.2023

José Álamos jose.alamos@haw-hamburg.de
Leandro Lanzieri leandro.lanzieri@haw-hamburg.de

Prof. Dr. Thomas C. Schmidt INET AG, Dept. Informatik HAW Hamburg t.schmidt@haw-hamburg.de

CoAP: Constrained Application Protocol

RESTful APIs for the IoT

Why do we need a web protocol for IoT?

- Web services on the Internet nowadays expose **RESTful APIs**
- **Avoid fragmentation** (silos) of IoT by:
    - Using and extending existing standard Web technologies
    - Providing standardized metadata
    - Integrating platforms, underlying protocols and application domains

Why do we need another web protocol?

- **HTTP does not fit** the constrained devices commonly found in the IoT:
    - Many 8-bit microcontrollers
    - Limited RAM and ROM
    - Battery-powered or severely energy constrained
    - Lossy wireless networks (e.g., 6LoWPAN)
    - Unreliable transports
    - Small link-layer frames

CoAP: Features

- Low header overhead and parsing complexity
- Supports URIs and Content-type
- Optional reliability (retries)
- Unicast and multicast requests
- Defined over multiple transports (including DTLS for security)

- For detailed information:
  - RFC 7252
  - https://coap.technology

REST model interactions

- **Servers** expose resources under URLs:

  `coap://node1.example.com/temperature`

- **Clients** operate on the resources utilizing methods:
  - GET
  - POST
  - PUT
  - DELETE

- The semantics of each method will ultimately depend on the specific application

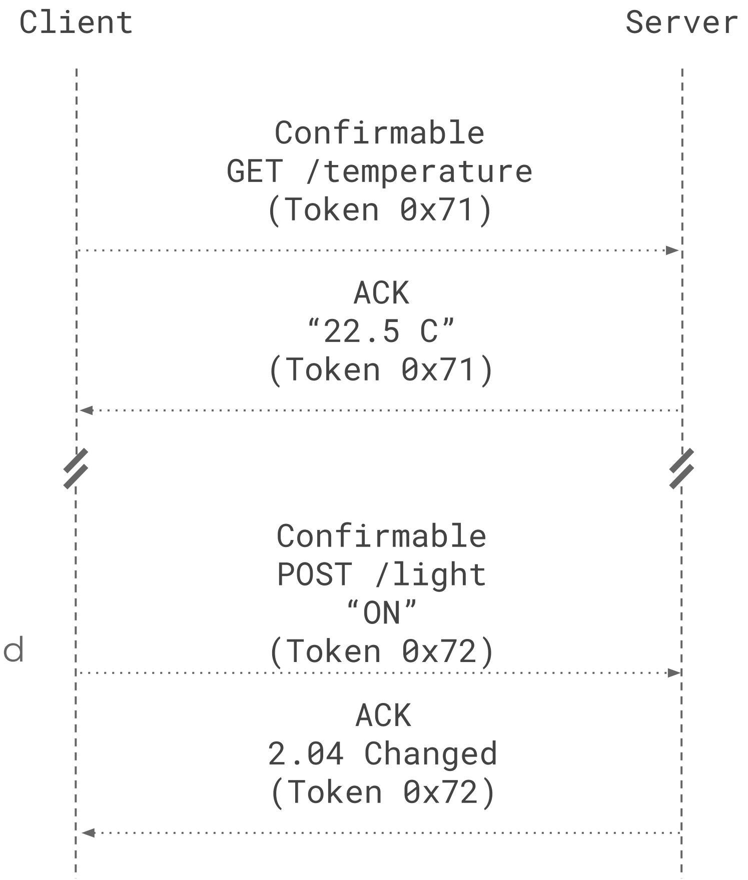
REST model interactions

- **Servers** expose resources under URLs:

  `coap://node1.example.com/temperature`

- **Clients** operate on the resources utilizing methods:
  - GET
  - POST
  - PUT
  - DELETE

- The semantics of each method will ultimately depend on the specific application

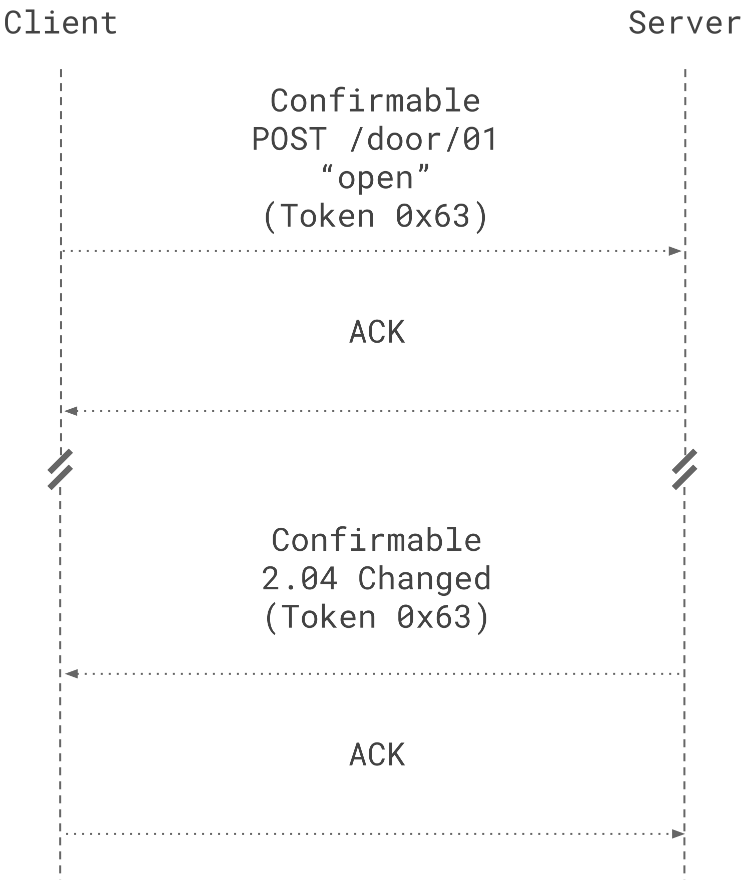


REST model interactions: separate response

- Server responses may be separate due to:
    - Long response processing time.
    - “Real-world” actions (e.g. switching a lock).
- Servers confirm requests by sending an ACK, and send responses at a later time, with a matching token.

REST model interactions: separate response

- Server responses may be separate due to:
  - Long response processing time.
  - “Real-world” actions (e.g. switching a lock).
- Servers confirm requests by sending an ACK, and send responses at a later time, with a matching token.

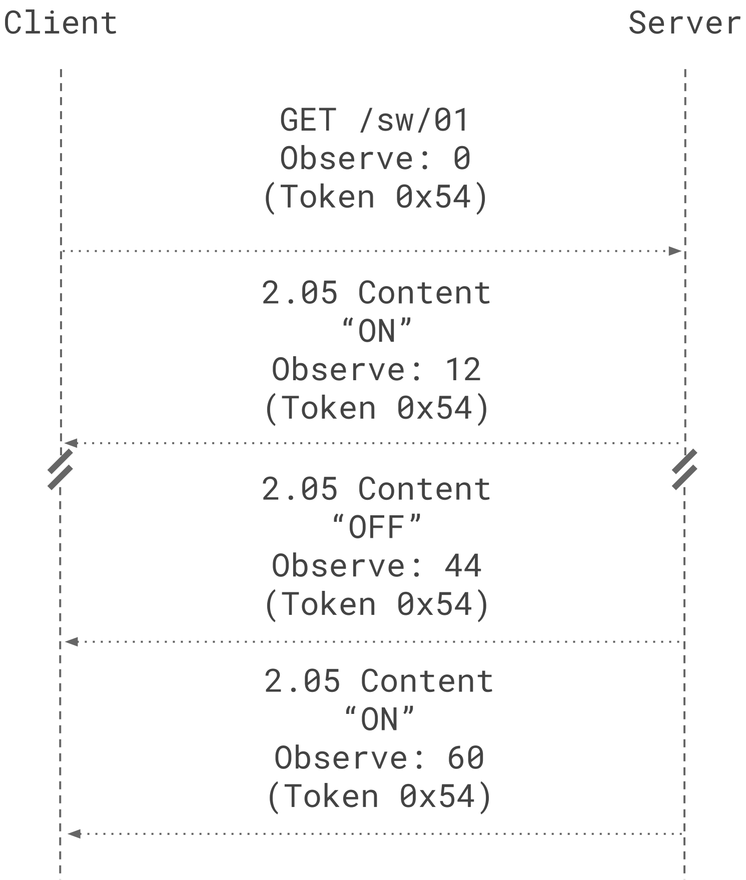


REST model interactions: observation

- Resources may change over time (e.g. the value of a light switch).
- Periodically polling resources consumes a lot of energy and bandwidth.
- The **observe** extension allows clients to request for notifications whenever the resource has changed (this is up to the server to determine).

REST model interactions: observation

- Resources may change over time (e.g. the value of a light switch).
- Periodically polling resources consumes a lot of energy and bandwidth.
- The **observe** extension allows clients to request for notifications whenever the resource has changed (this is up to the server to determine).



Resource discovery: /.well-known/core

- Clients can discover which resources a given server provides
- The interface accepts GET requests, and returns a list of resources in LinkFormat:

```
Client Request:
   GET /.well-known/core

Server Response:
   2.05 Content
   </sensors/temp>;if="sensor",
   </sensors/light>;if="sensor"
```

URIs　　Attributes

Resource discovery: /.well-known/core

- Clients can discover which resources a given server provides
- The interface accepts GET requests, and returns a list of resources in LinkFormat:

```
Client Request:
   GET /.well-known/core

Server Response:
   2.05 Content
   </sensors/temp>;if="sensor",
   </sensors/light>;if="sensor"
```

URIS **Attributes**

- Query filter parameters can be added, when a resource with specific metadata is required:

Filter by resource type

```
Request: GET /.well-known/core?rt=light-lux

Response: 2.05 Content
 </sensors/light>;rt="light-lux";if="sensor"
```

Resource discovery: resource directory

- In some scenarios direct discovery of resources may not be possible
  - Long-sleeping nodes
  - Multicasting not efficient
- Resource Directories (RD) contain information about resources in other servers
- A Resource Directory has two interfaces
  - Registration interface: servers register their resources
  - Lookup interface: clients look for resources exposed by servers

Resource discovery: resource directory

Operation flow

1. The server finds the RD
    - Statically configured
    - Discovery procedure (e.g. multicast)
2. The server **registers** itself on the RD by sending information about its resources
    - The server may periodically update the registration
3. The client performs a **lookup** on the RD, to find a resource with specific characteristics
    - It may use the observe mechanism to be notified about new resources

Resource discovery: resource directory

1. A server finds the RD (may be static or via discovery)
2. The server registers, and sends information about its resources

```
Request:
     POST coap://rd.example.com/rd?ep=node1
     Content-Format: 40
     Payload:
          </sensors/temp>;rt=temperature-c;if=sensor

Response:
     2.01 Created
     Location-Path: /rd/4521
```

Resource discovery: resource directory

3. The server may periodically update the registration
4. A client performs a lookup on the RD, to find a resource with specific characteristics

```
Request:
     GET /rd-lookup/res?rt=tag:example.org,2020:temperature

Response:
     2.05 Content
     Payload:
     <coap://[2001:db8:3::123]:61616/temp>; rt="tag:example.org,2020:temperature"
```

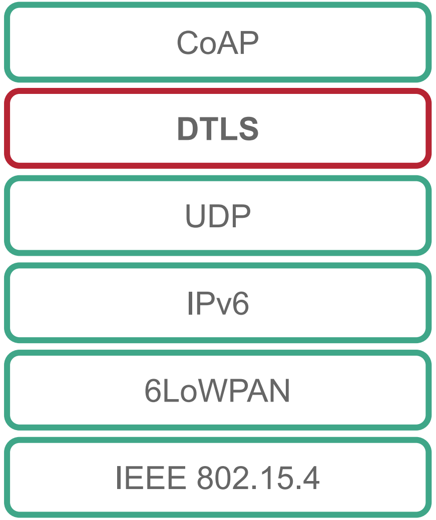
Resource discovery: resource directory

The client can even take advantage of the observe mechanism, to be notified about newly registered nodes

```
Request:
     GET /rd-lookup/res?rt=tag:example.org,2020:light
     Observe: 0

Response:
     2.05 Content
     Observe: 23
     Payload: empty

(at a later point in time…)

Response:
     2.05 Content
     Observe: 24
     Payload:
     <coap://[2001:db8:3::124]/west>;rt="tag:example.org,2020:light",
     <coap://[2001:db8:3::124]/south>;rt="tag:example.org,2020:light",
     <coap://[2001:db8:3::124]/east>;rt="tag:example.org,2020:light"
```

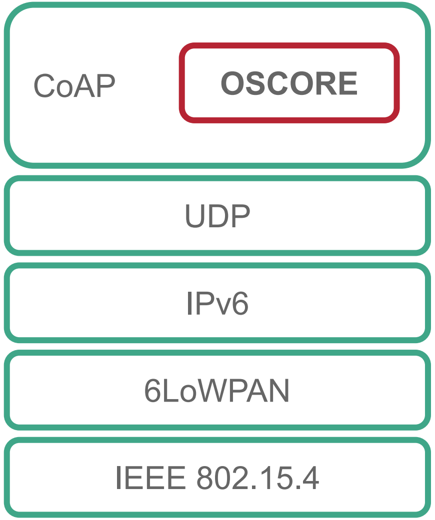
Securing CoAP: DTLS

- Datagram Transport Layer Security
  - Four different modes
    - NoSec: no protocol-level security
    - PreSharedKey: Symmetric keys
    - RawPublicKey: Asymmetric keys
    - Certificate: Asymmetric keys with X.509 certs.
  - Nodes establish a point-to-point DTLS session
    - Provides authentication, integrity, and confidentiality
    - Intermediate nodes (e.g., gateways) need to decrypt and re-encrypt
      - Difficult to cache
      - Difficult to proxy



Securing CoAP: OSCORE

- Object Security for Constrained RESTful Environments
  - Uses pre-shared keys
  - Security at object level (no point-to-point session)
    - The original CoAP message is encrypted and encapsulated as a COSE object (CBOR Object Signing and Encryption)
    - The encapsulated message is nested in an outer CoAP message
    - Provides integrity, authenticity, and confidentiality **at CoAP level**
    - Allows protecting multicast messages
    - Allows caching and proxies



Questions?