Communication Technologies for the IoT

Networking tiny devices

Who Needs to Communicate in IoT?

IoT

- Wearable
- Environmental sensor/actuators
- Domotic sensors/actuators
- White goods
- Vehicles

IIoT

- Higher Layer components among themselves
- Controllers with higher layer components (MES, ERP, Data Bases, Data Lakes)
- Controllers with controllers (PLCs, SCADA)
- Sensors with controllers (PLCs, DCS)

Connectivity in I4.0 – Why is it so different?

Connectivity

- A different environment
  - Strong electromagnetic interference
  - Harsh conditions (temperature, humidity, etc.)
  - High mechanical stress
- Different Requirements
  - Timeliness
  - Determinism
  - Reliability

- IT/OT Check list:
  - What is the network size
  - Mobile or Fixed Nodes
  - Any service QoS constraint (throughput, latency)?
  - The “integration issue”
  - Any budget?

What is the traffic in IoT?

- ***Cyclic Traffic***: periodic transfer of sensor measurements and set points

  Example: velocity control loop of a electrical engine requires sending a set point (2 bytes) every 100us

- ***Acyclic Traffic***: traffic generated by unpredictable events

  Example: alarm triggered by a failing component of the industrial process

- ***Multimedia Traffic***: images/video

  Example: intrusion detection systems, defect inspection

- ***Backhand Traffic:*** *aggregated flows*

IoT KPIs - Throughput

- *Throughput (rate)*: number of bits (bytes) per second that can be sent across a link (Unit measure: [bit/s] or [byte/s])

- Depends on: link type

1 **kbps** (kb/s) = 10^3 bps

1 **Mbps** (Mb/s) = 10^6 bps

1 **Gbps** (Gb/s) = 10^9 bps

1 **B** = 8 bit

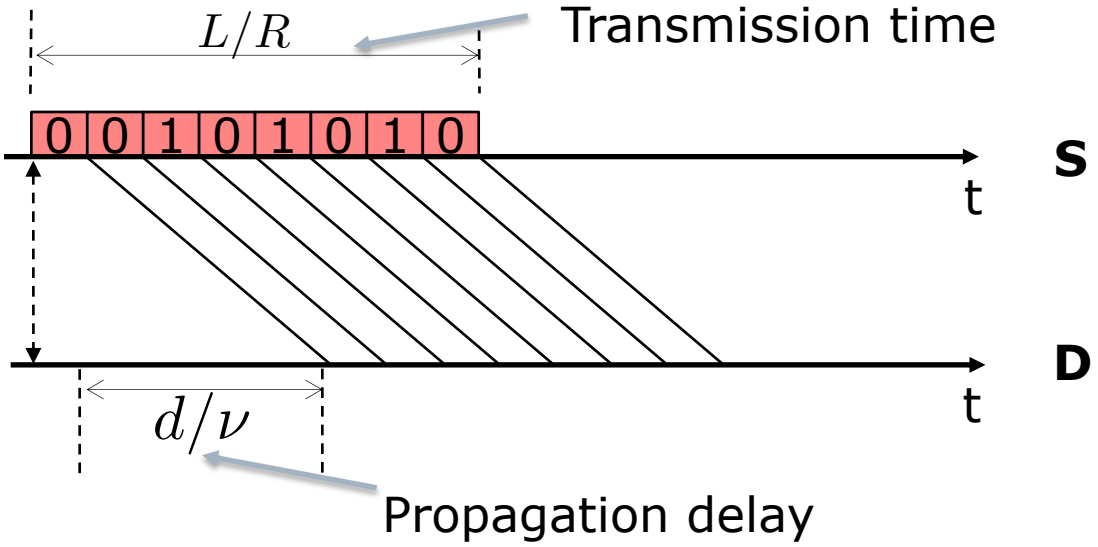
1 **kB** = 10^3 B

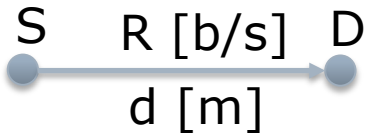
1 **MB** = 10^6 B

1 **GB** = 10^9 B

IoT KPIs - Throughput – Delay KPIs

- *Delivery Time*: time necessary to send one *service data unit* from source to destination (Unit measure: [s])
- Depends on: transmission time, propagation time, protocol execution time
- *Example*: link *d* meters long with a throughput *R,* source *S* to send *L* bits to *D,* wave propagation speed



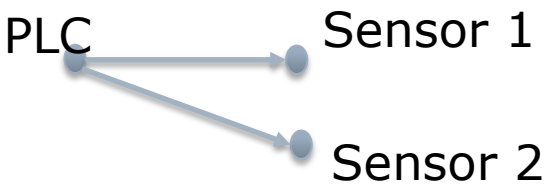


$$T = L/R + d/\nu$$

IoT KPIs – Delay KPIs

- *Minimum Cycle Time (MCT):* minimum time required to execute a cycle in a control loop

Example: one controller, 2 sensors; control loop: PLC polls sensor 1, sensor 1 sends back reading, PLC does the same for sensor 2



$$MCT = T_{PLC-S1} + T_{S1-PLC} + T_{PLC-S2} + T_{S2-PLC}$$

IoT KPIs - Precision KPIs

- *Jitter:* precision/reliability in performing periodic operations

Example:

nominal required time to execute a generic control operation, *P*; actual current time to execute the generic operation $\hat{P}$

$$J = \frac{|P - \hat{P}|}{P}$$

A primer on networking fundamentals

How to Define a Communication Technology?

- Physical infrastructure: hosts (sensors/actuators, PLC, SCADA), links (fiber optics, wireless, wired), network nodes (access point, switch, router, gateway)
- Network architectures/topologies
- A set of communication protocols

Physical Components

- Hosts: *traffic end points*

Physical/virtual servers





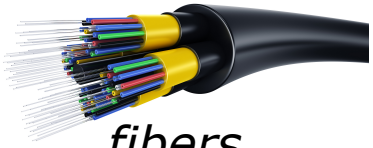
Personal devices





Smart objects

- Links



fibers

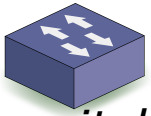
Twisted pairs

Coax cables

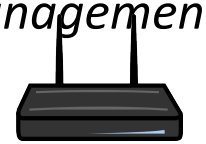
wireless

- Network devices: *interconnection and network management*

router

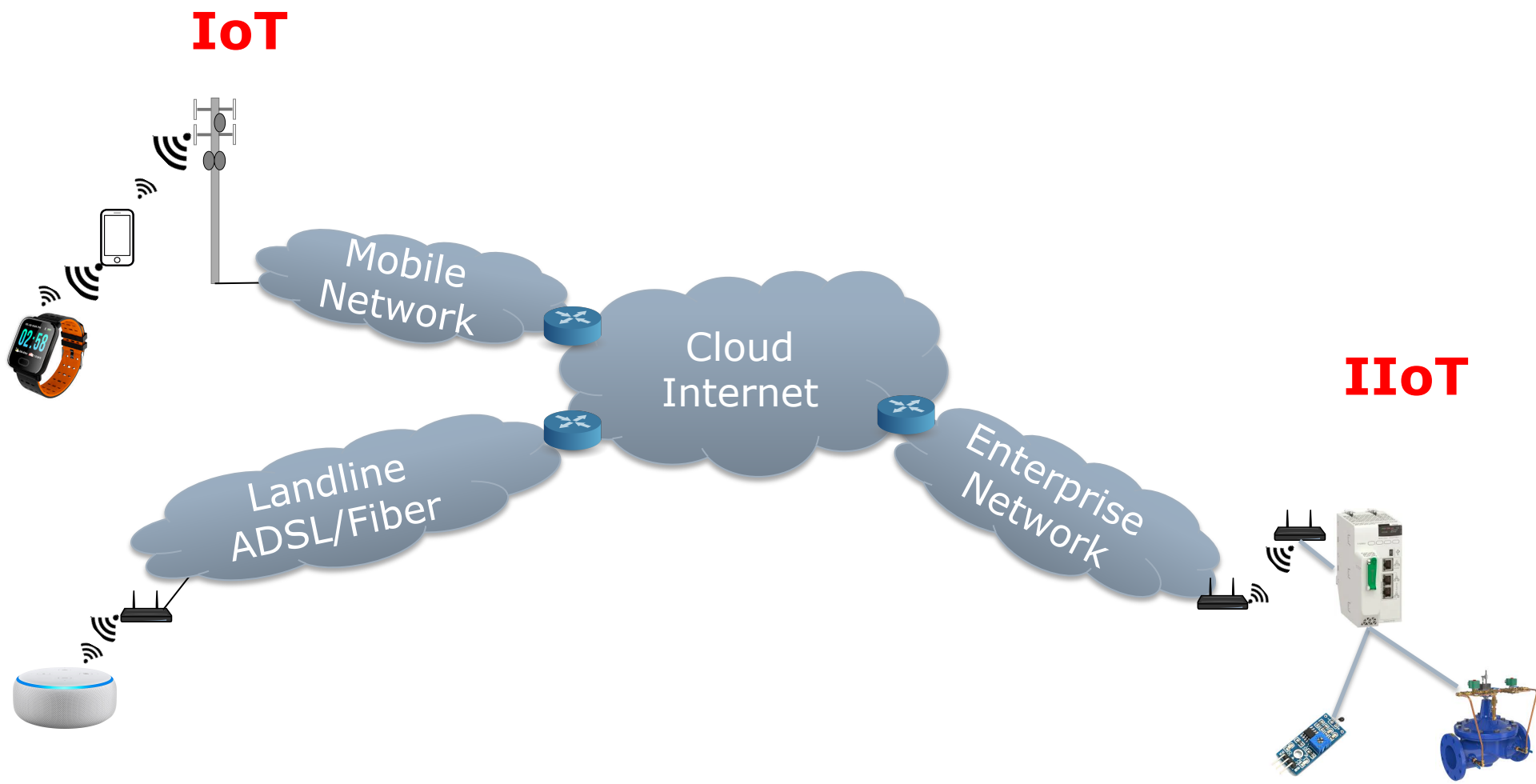


switch

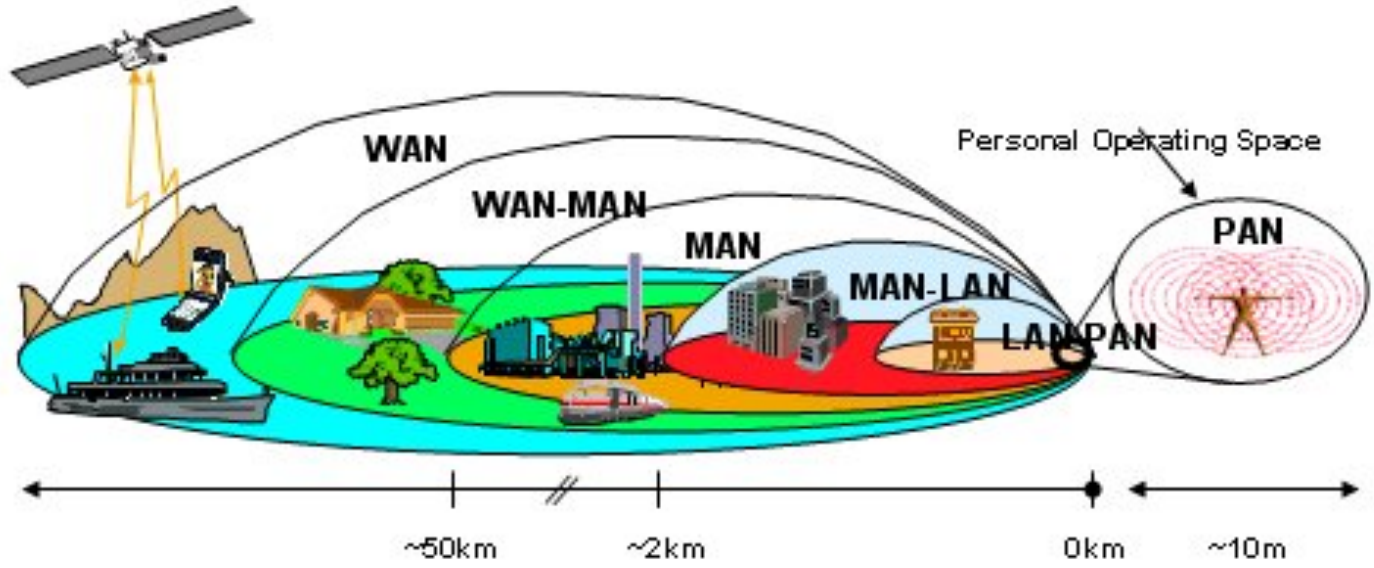


access point

IoT Connectivity: Never (seldom) one single-hop/technology

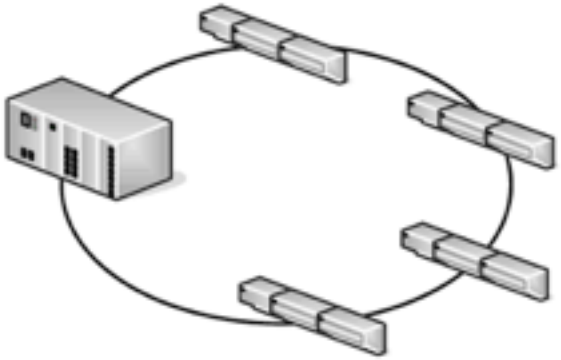


Comm Tech Classification: the Range Matters

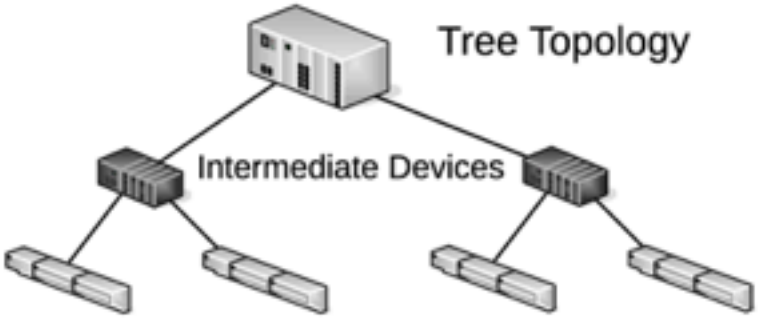


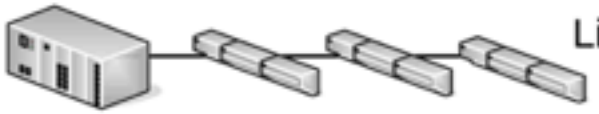
The IoT will include/leverage heterogeneous technologies

Comm Tech Classification: the Shape Matters

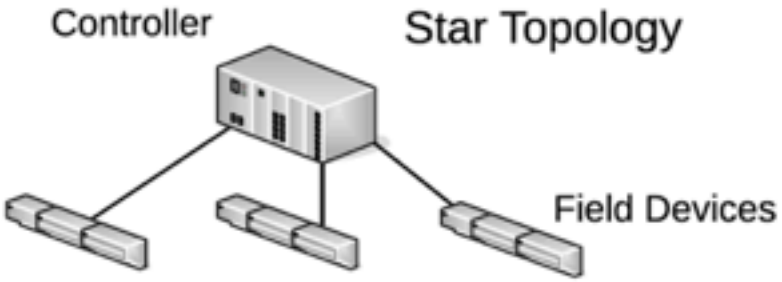


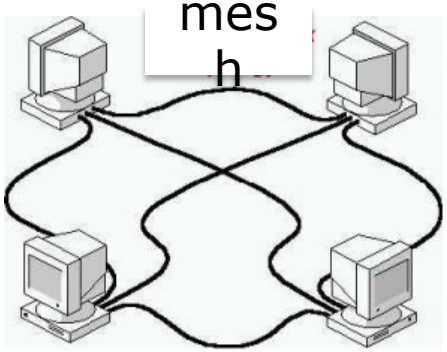
Ring Topology



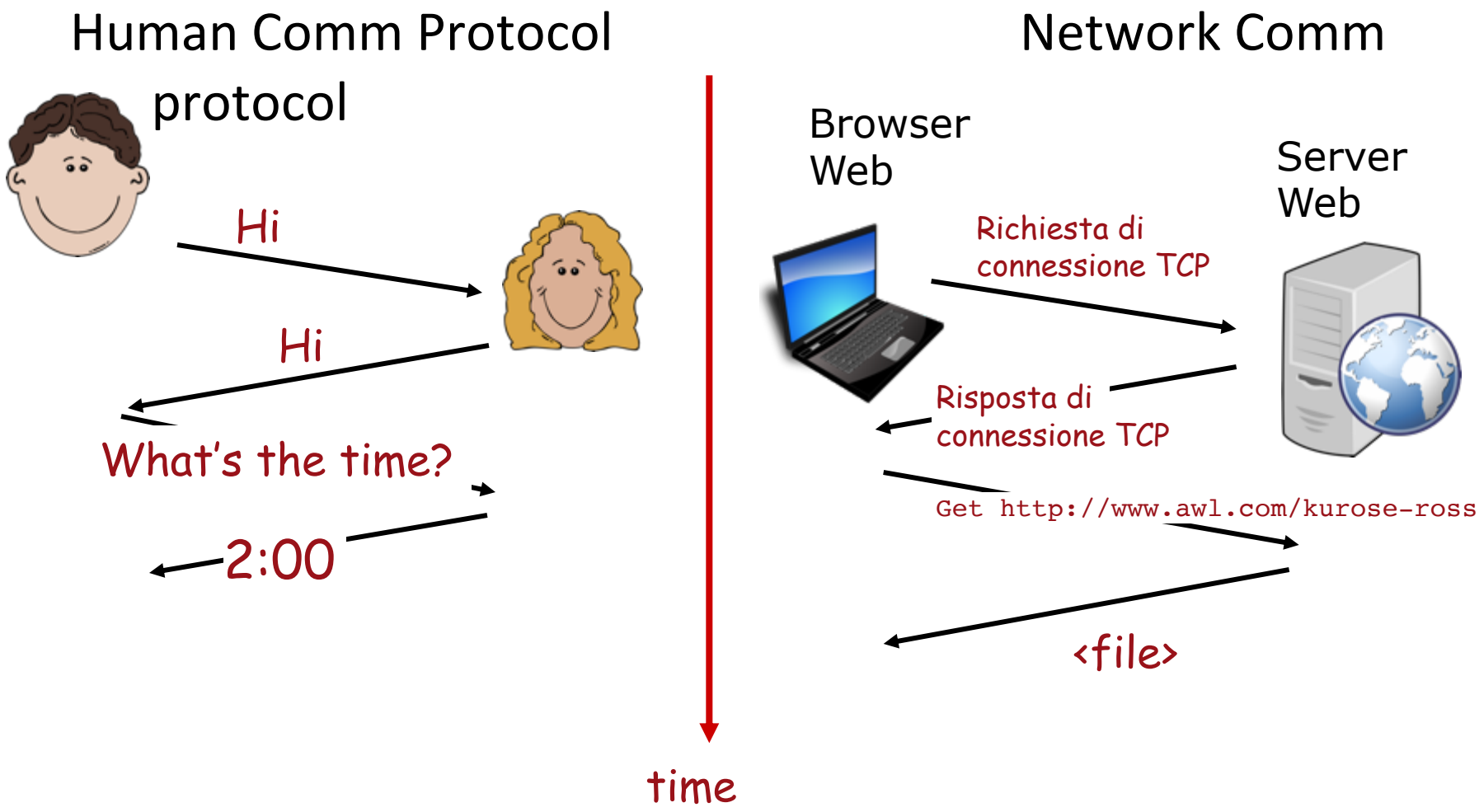


Linear Chain Topology





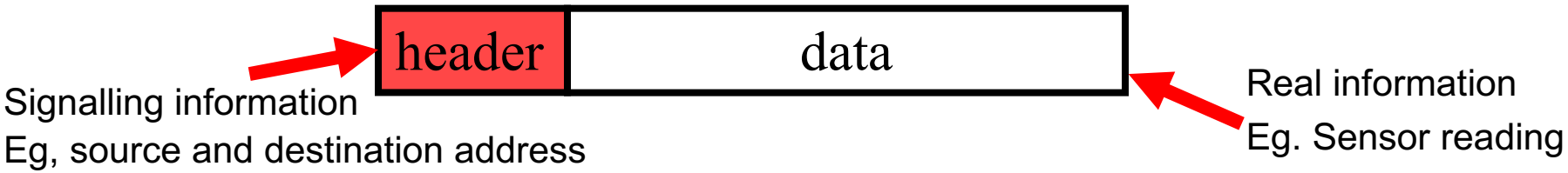
Communication Protocols



Communication Protocols

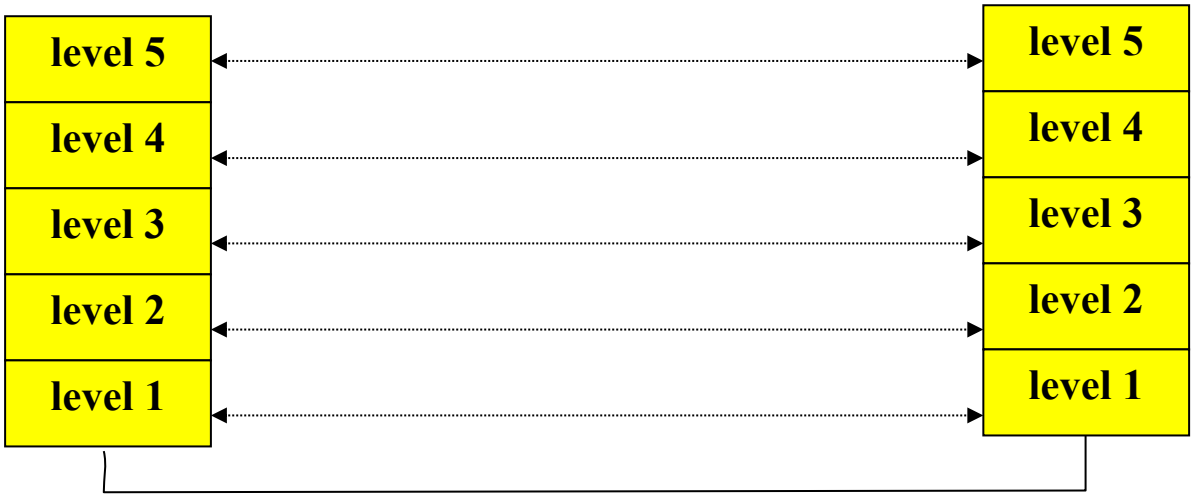
- Set of rules underpinning the exchange of information between two entities
  - Message format
  - Connection set up procedures
  - Connection tear down procedures
  - Semantics

- IoT comm networks protocols exchange packets (like the Internet)

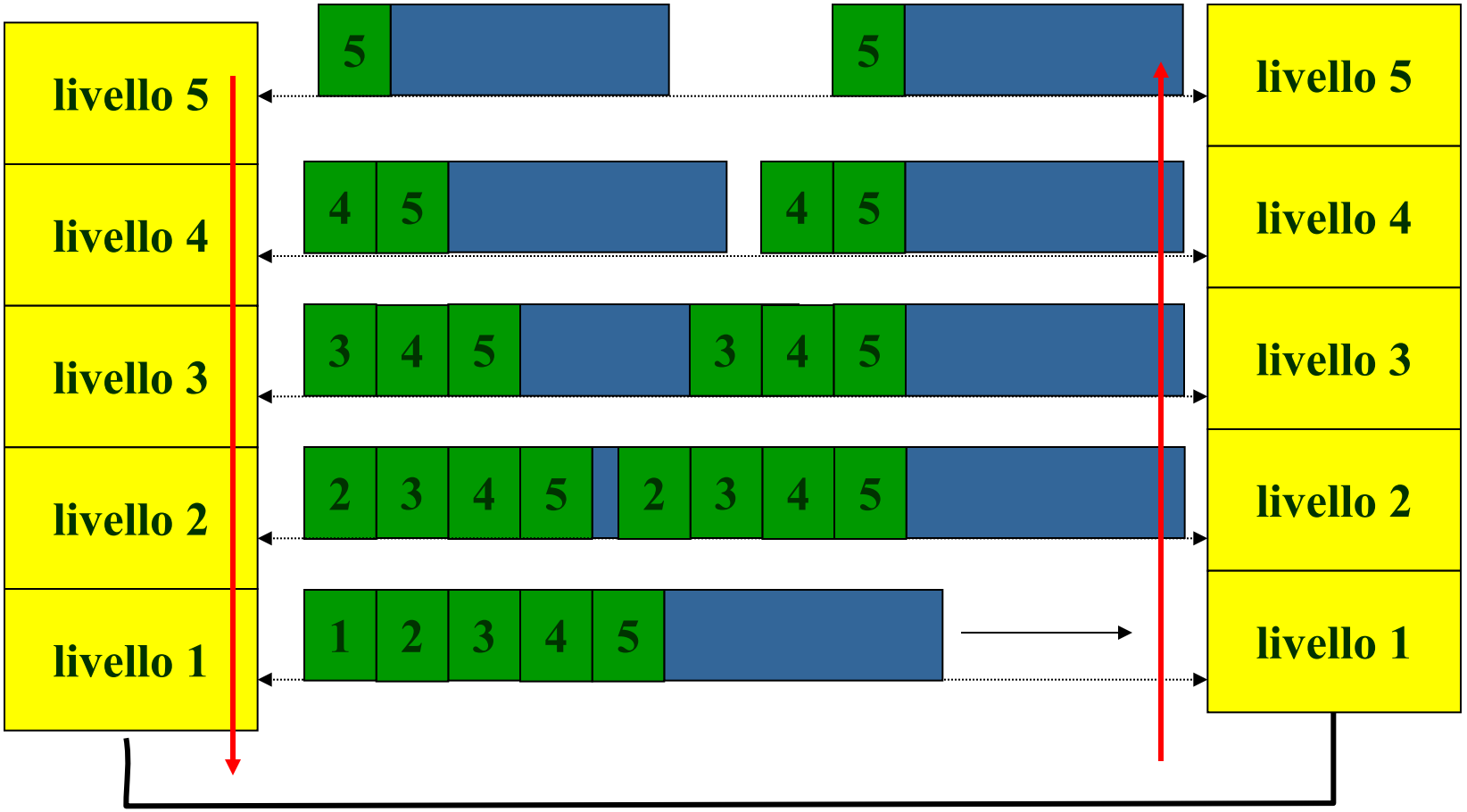


Layered Architecture

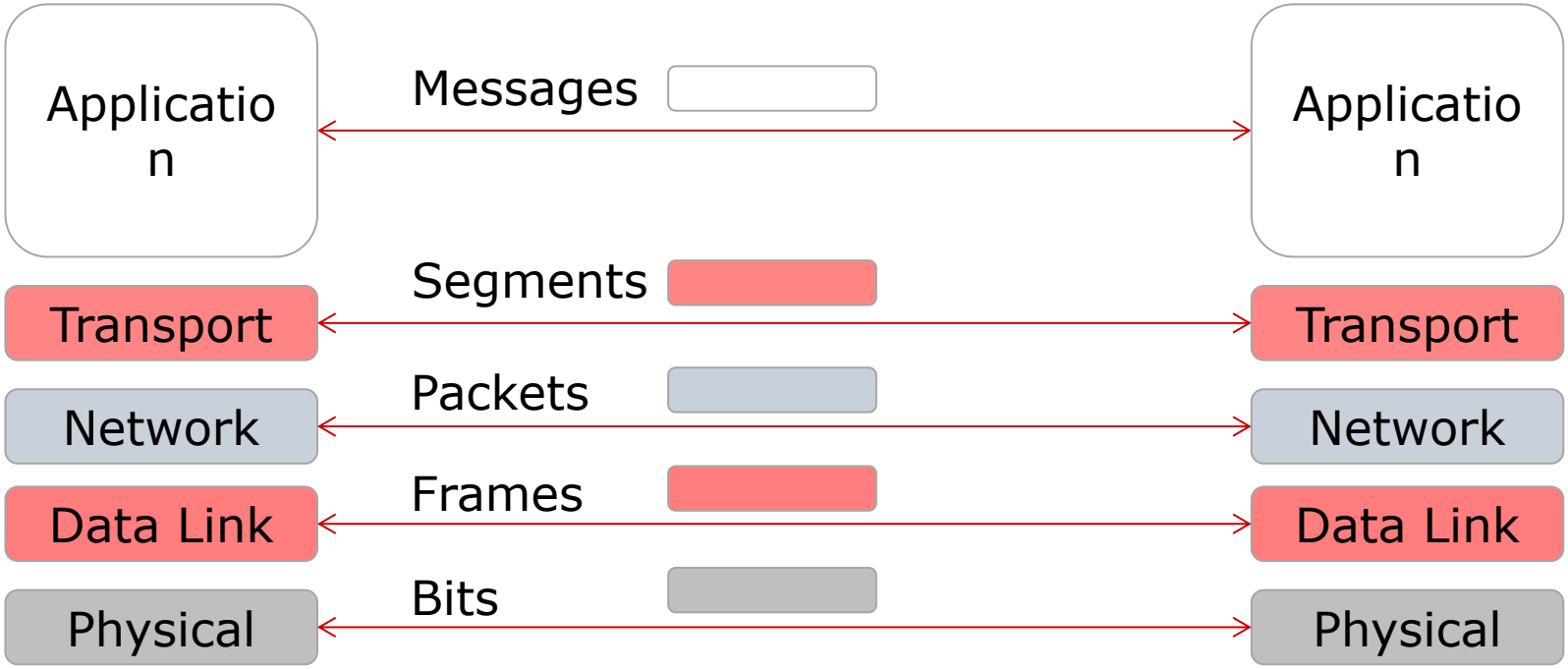
- Communication protocols are commonly layered
- Each layer provides some services/funstionalities to upper layers



Layered Architecture



Internet Protocol Stack

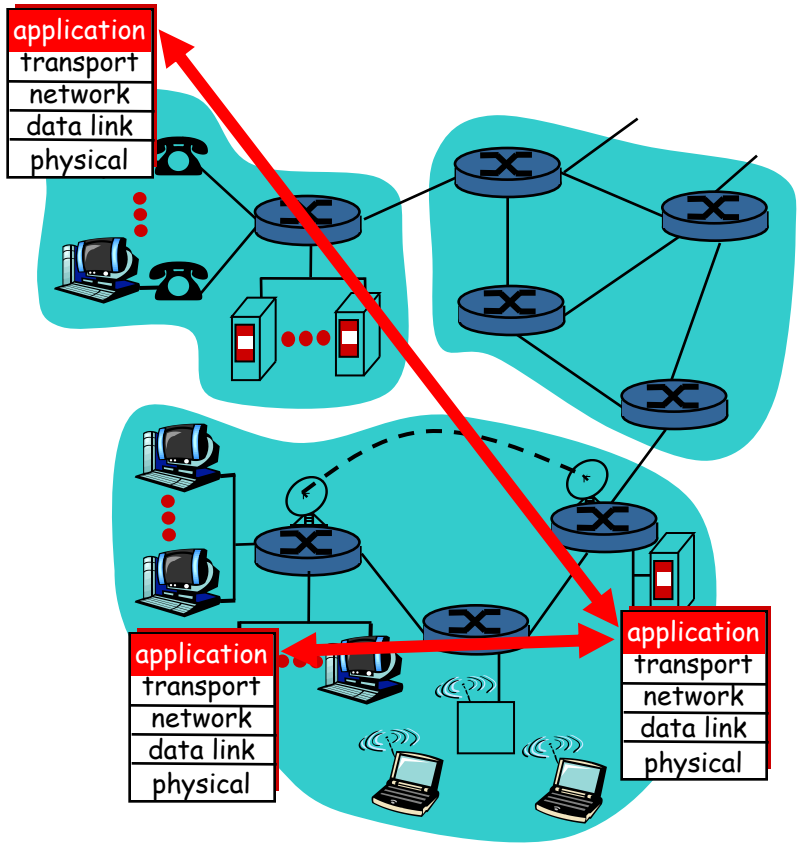


Application Layer Protocols for the IoT

What is an application?

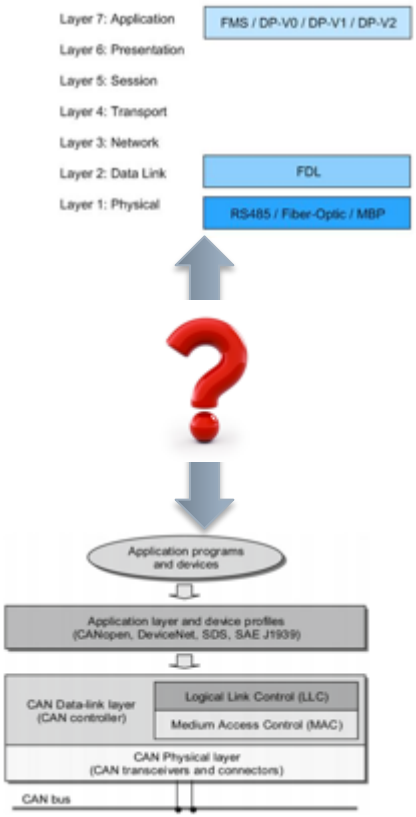
- **A program (piece of software)**:
  - In execution at a host
  - Able to communicate with another program through a network

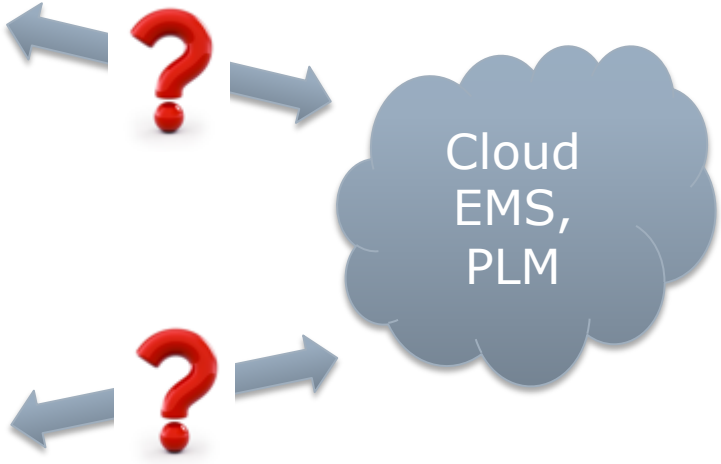
- **Example:** the web browser (*FireFox, Safari, Chrome*, ecc..) is a software which communicates with another software called *web server* (www.google.com, www.amazon.com, ecc..)



Need for Mediation

- Communication Technologies in I4.0 are highly fragmented
- Different Technologies (languages) cohexist at the factory
- Field/Factory Technologies do not «speak Internet»





Interoperability

Solution: Service Oriented Architectures

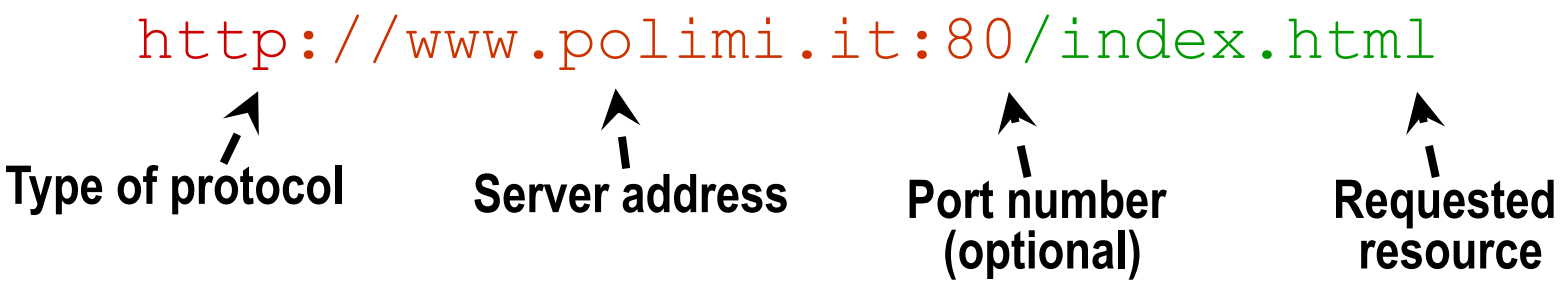
Application Layer Protocols

- Client/Server «web browsing approach»
  - HTTP
  - COAP
  - OPC-UA
- Publish/Subscribe «Twitter approach»
  - MQTT
  - OPC-UA

Hyper Text Transfer Protocol

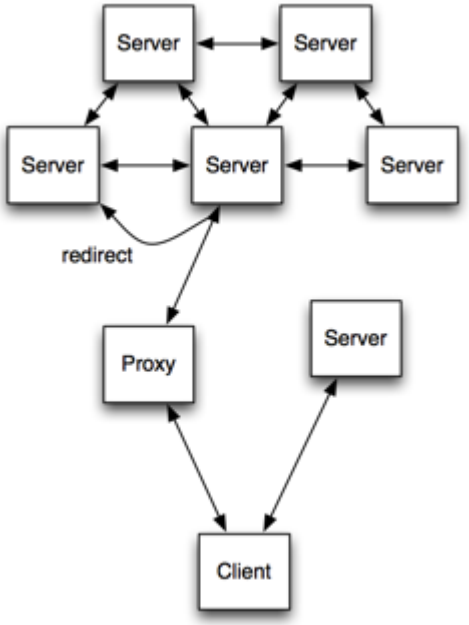
HTTP messages – The Web Pages

- Primer:
  - *Web pages* composed of *resources (HTML, images, applet, …)*
  - each *resource* has a *Uniform Resource Locator (URL) or Uniform Resource Identifier (URI)*

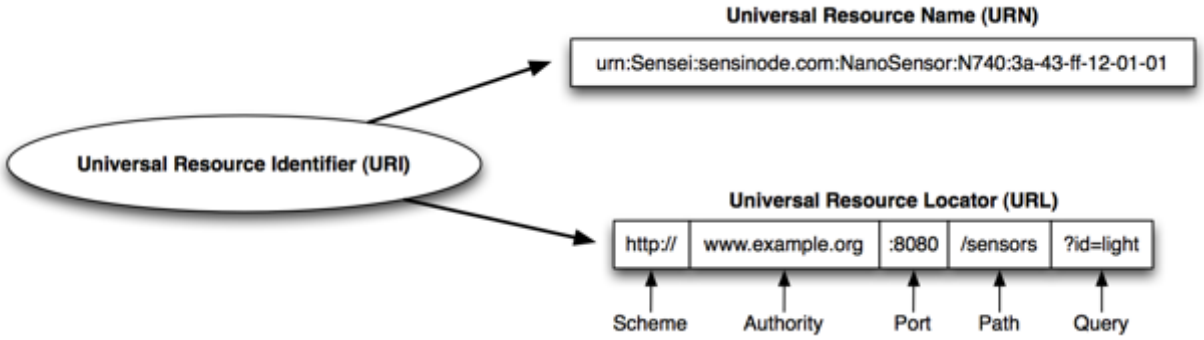


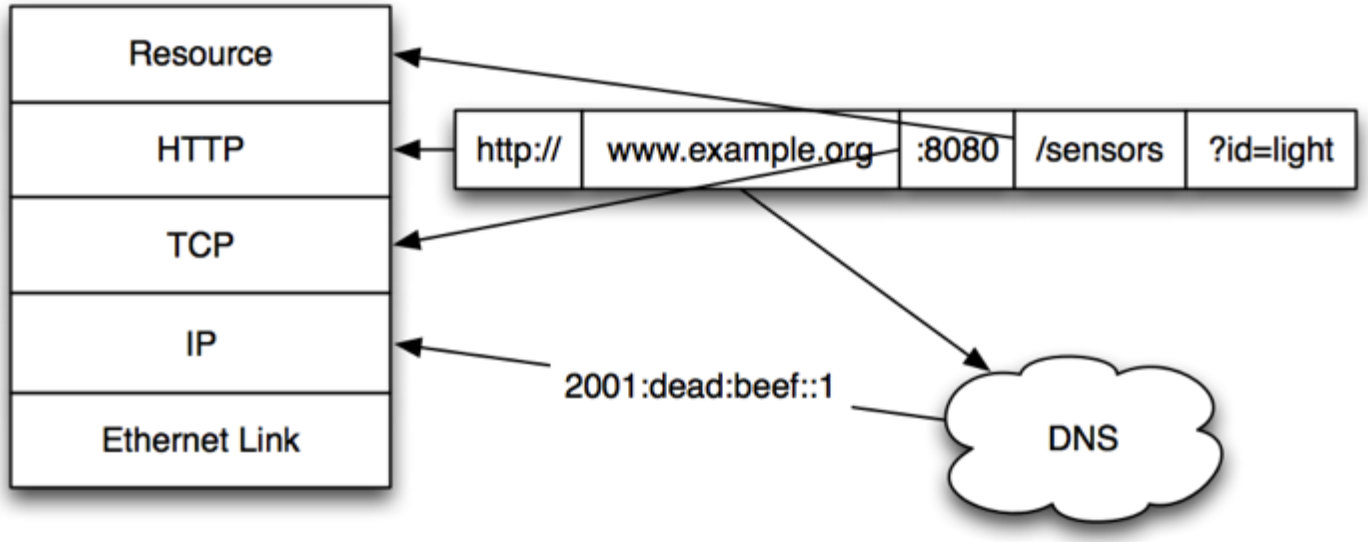
How Does the Web Work?

- Resources in the Web are:
  - managed by servers
  - identified by URIs
  - accessed synchronously by clients through request/response paradigms
- In a word, Representational State Transfer (REST)



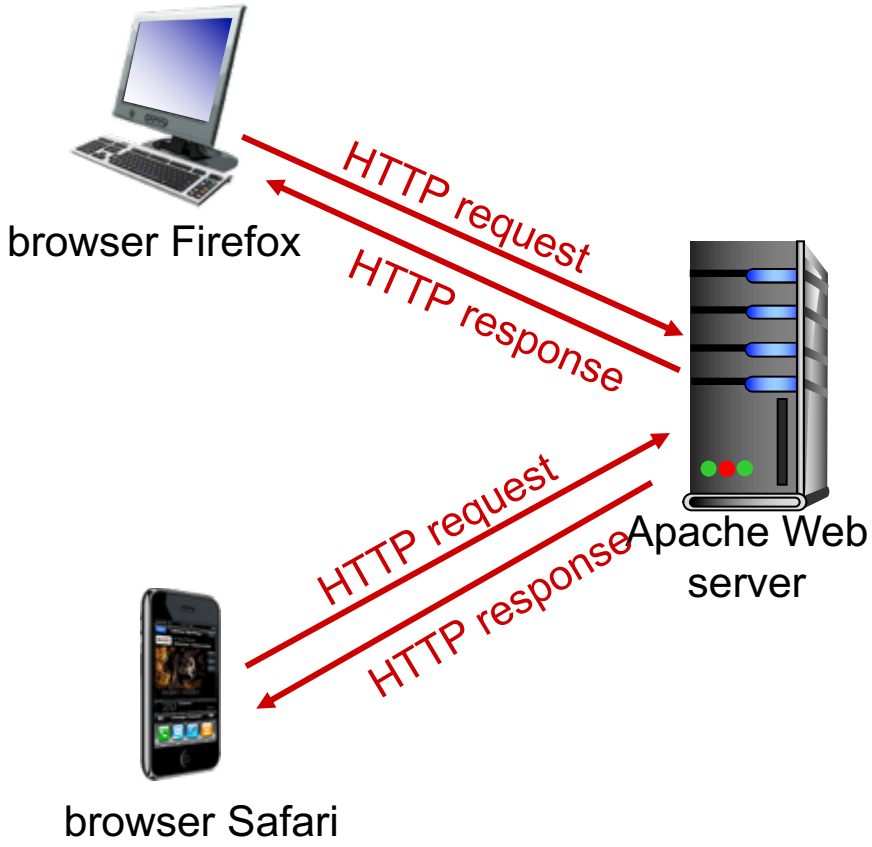
URL Resolution





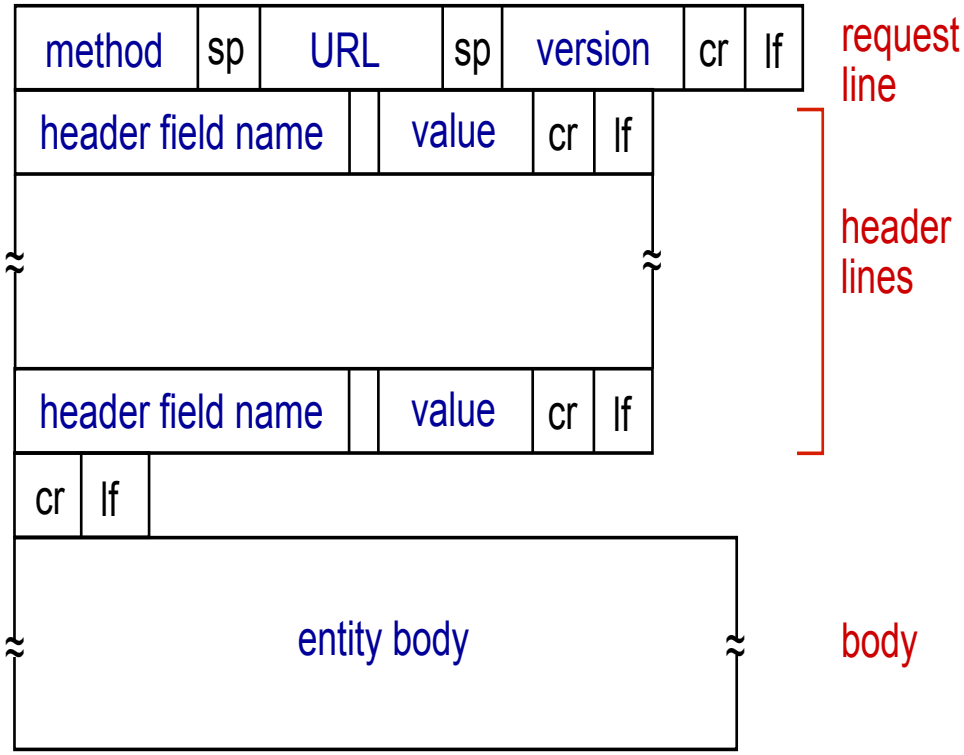
HTTP Communication

- *client/server* architecture*:*
  - *client*: issues HTTP requests for specific resources (URIs)
  - *server*: answers backs with the required resource
- Transfer is stateless: no memory maintained at the server



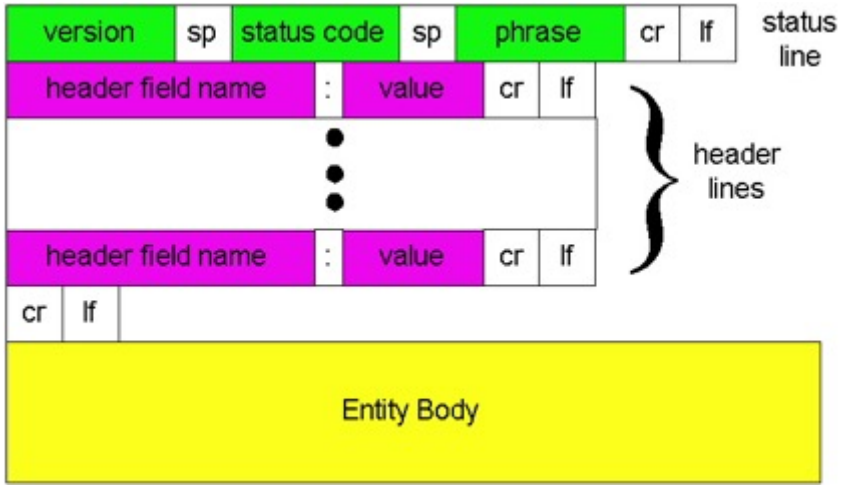
HTTP Requests

- ASCII encoding (*human-readable*)




```
GET /index.html HTTP/1.1\r\n
Host: www-net.cs.umass.edu\r\n
User-Agent: Firefox/3.6.10\r\n
Accept: text/html,application/xhtml+xml\r\n
Accept-Language: en-us,en;q=0.5\r\n
Accept-Encoding: gzip,deflate\r\n
Accept-Charset: ISO-8859-1,utf-8;q=0.7\r\n
Keep-Alive: 115\r\n
Connection: keep-alive\r\n
\r\n
```

HTTP Responses




```
HTTP/1.1 200 OK\r\n
Date: Sun, 26 Sep 2010 20:09:20 GMT\r\n
Server: Apache/2.0.52 (CentOS)\r\n
Last-Modified: Tue, 30 Oct 2007 17:00:02 GMT\r\n
ETag: "17dc6-a5c-bf716880"\r\n
Accept-Ranges: bytes\r\n
Content-Length: 2652\r\n
Keep-Alive: timeout=10, max=100\r\n
Connection: Keep-Alive\r\n
Content-Type: text/html; charset=ISO-8859-1\r\n
\r\n
data data data data data ...
```

Status code:

- 1xx: information
- 2xx: success
- 3xx: redirection
- 4xx: client-side error
- 5xx: server-side error errore lato server

[1]complete list in RFC 2616
http://www.w3.org/Protocols/rfc2616/rfc2616-sec10.html

COnstrained Application Protocol (COAP)

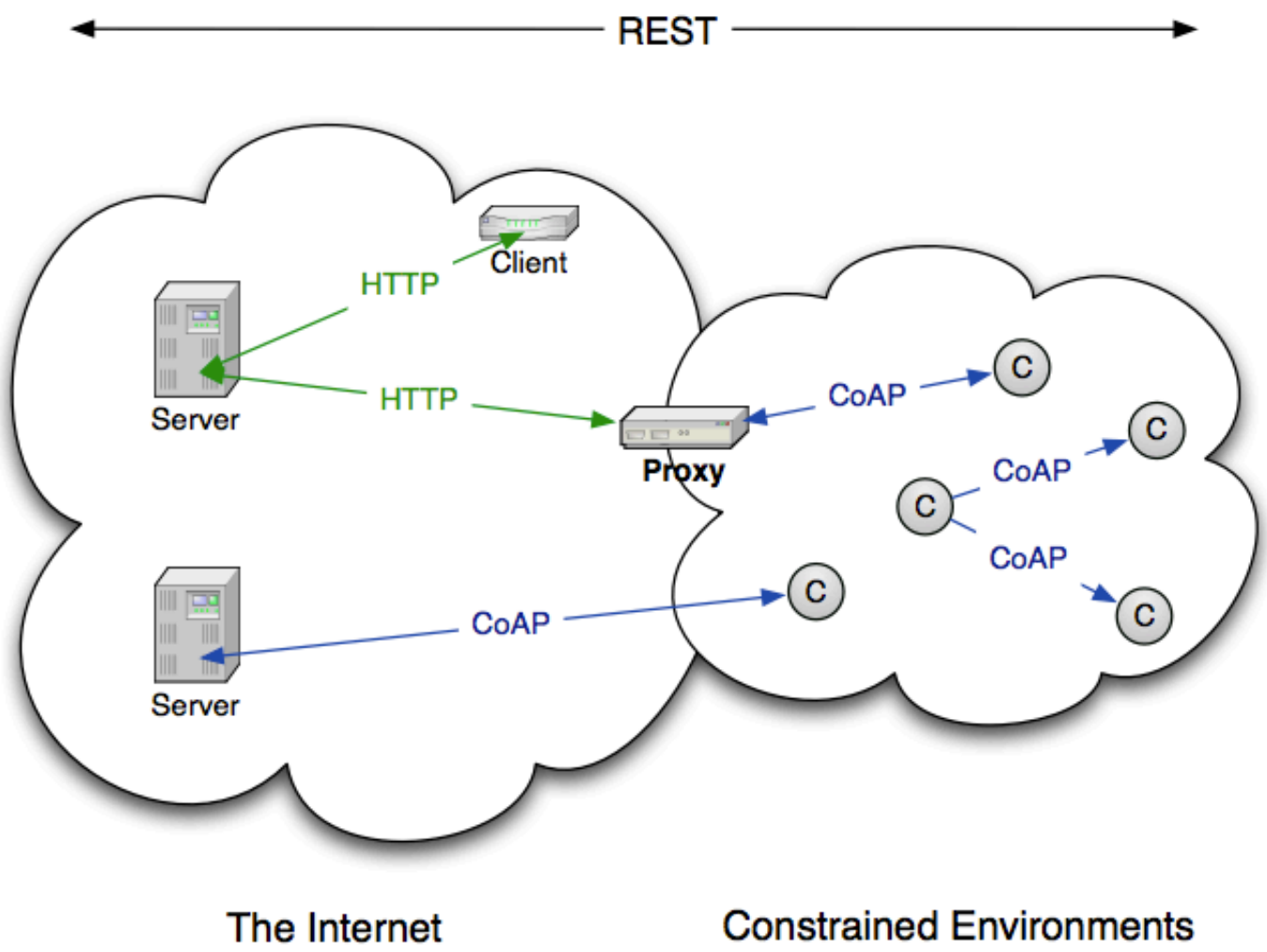
Background

- GOAL: to enable web-based services in constrained wireless networks
  - 8 bit micro-controllers
  - limited memory
  - low-power networks
- Problem: WEB solution are hardly applicable
- Solution: re-design web-based services for constrained networks -> COAP

CoAP At a Glance

- Embedded web transfer protocol (coap://)
- Asynchronous transaction model
- UDP binding with reliability and multicast support
- GET, POST, PUT, DELETE methods
- URI support
- 4 byte header
- Subset of MIME types and HTTP response codes
- Built-in discovery
- Optional observation and block transfer

The CoAP Architecture

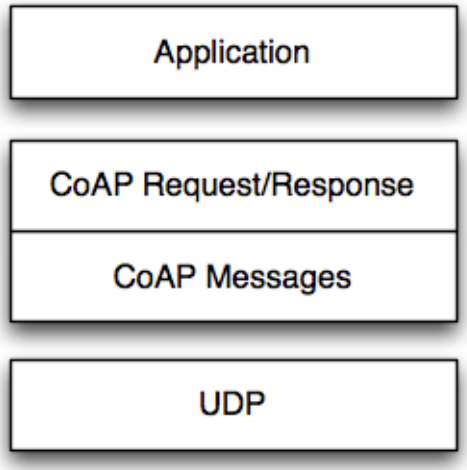


COAP Messaging Basics

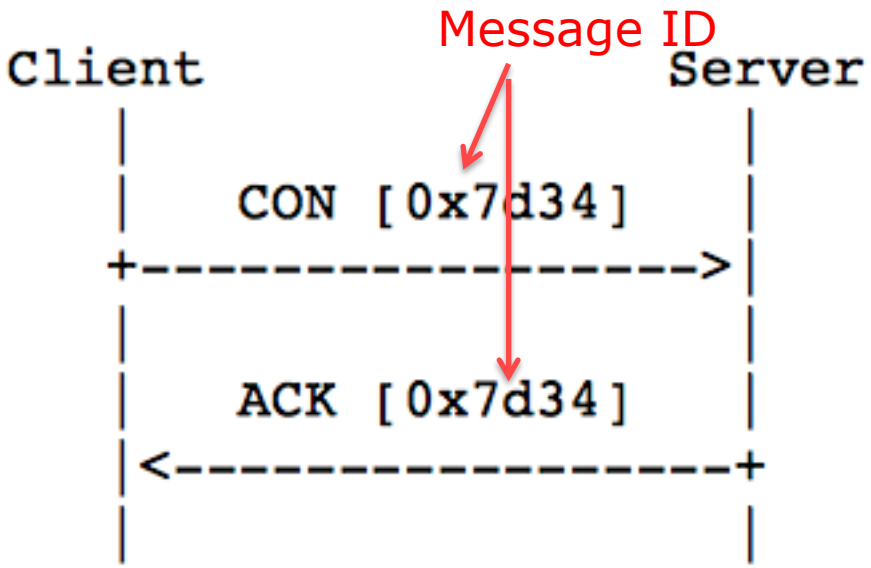
- Transport:
  - (mainly) UDP binding
- Message Exchange between Endpoints
  - Messages with 4 bytes header (shared by request and responses) containing a message ID (16 bits)
  - Reliable exchange through Confirmable Messages which must be acknowledged (through ACK or Reset Messages). Simple Stop-and-Wait retransmission with exponential back-off.
  - Unreliable exchange through Non-Confirmable Message
  - Duplicate detection for both confirmable and non-confirmable messages (through message ID)

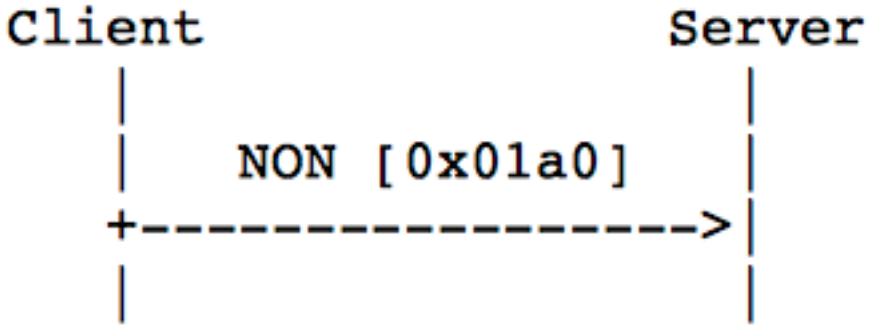
COAP Message Semantics

- REST Request/Response piggybacked on CoAP Messages
- Method, Response Code and Options (URI, content-type etc.)

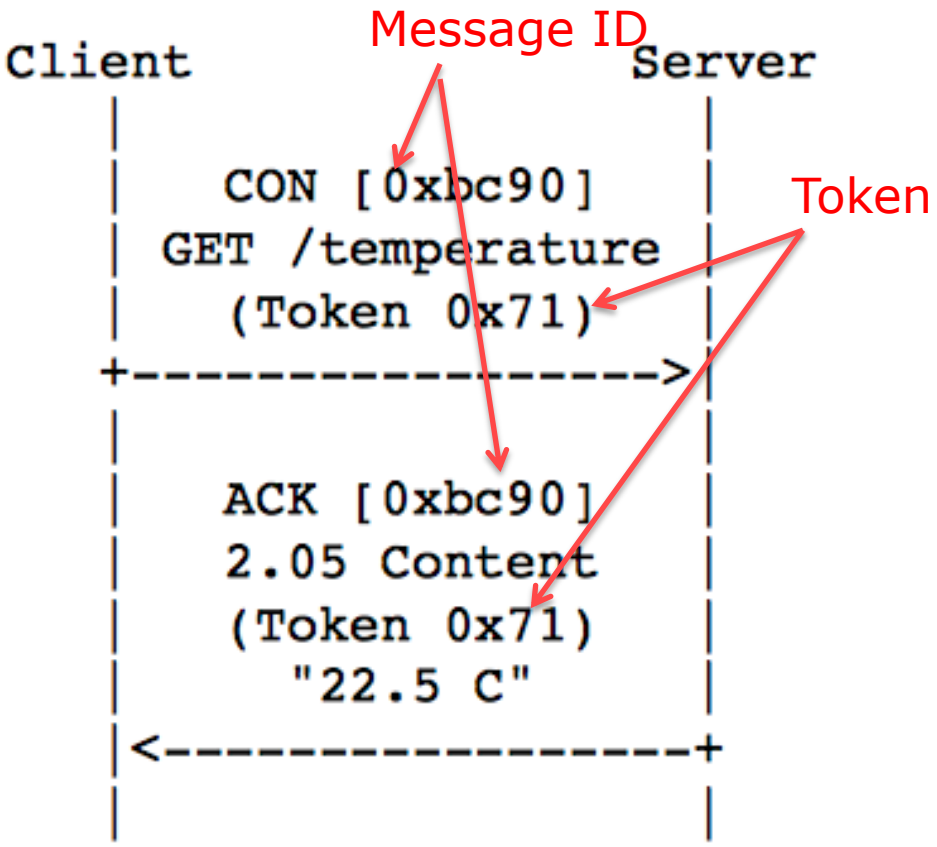


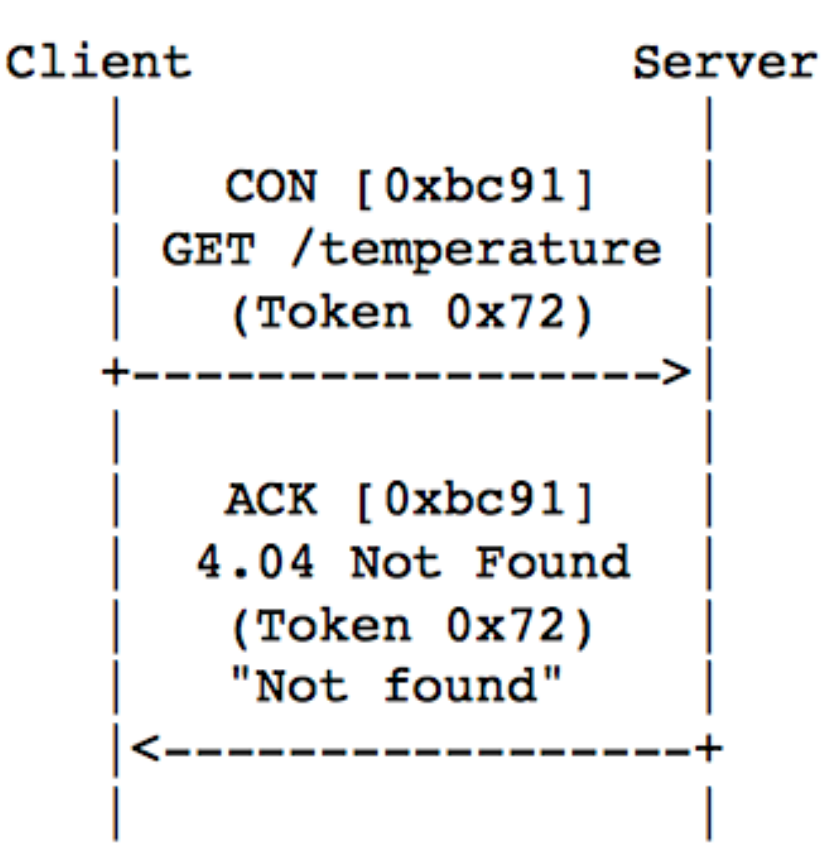
COAP Messaging



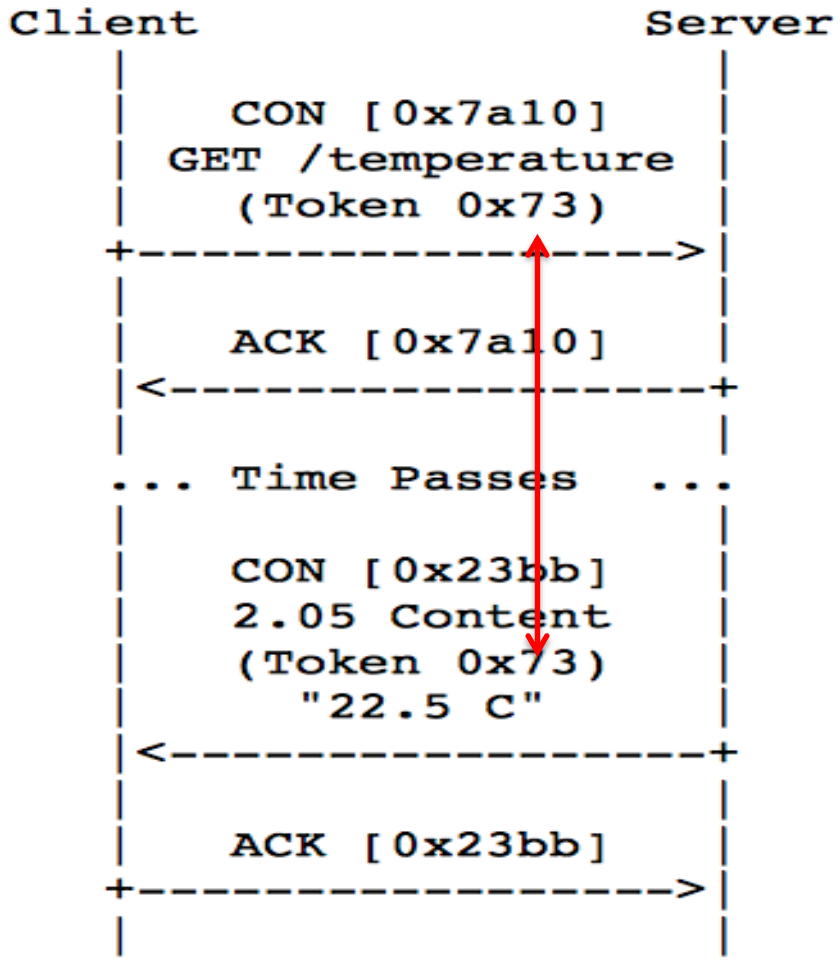


COAP Request/Response Examples

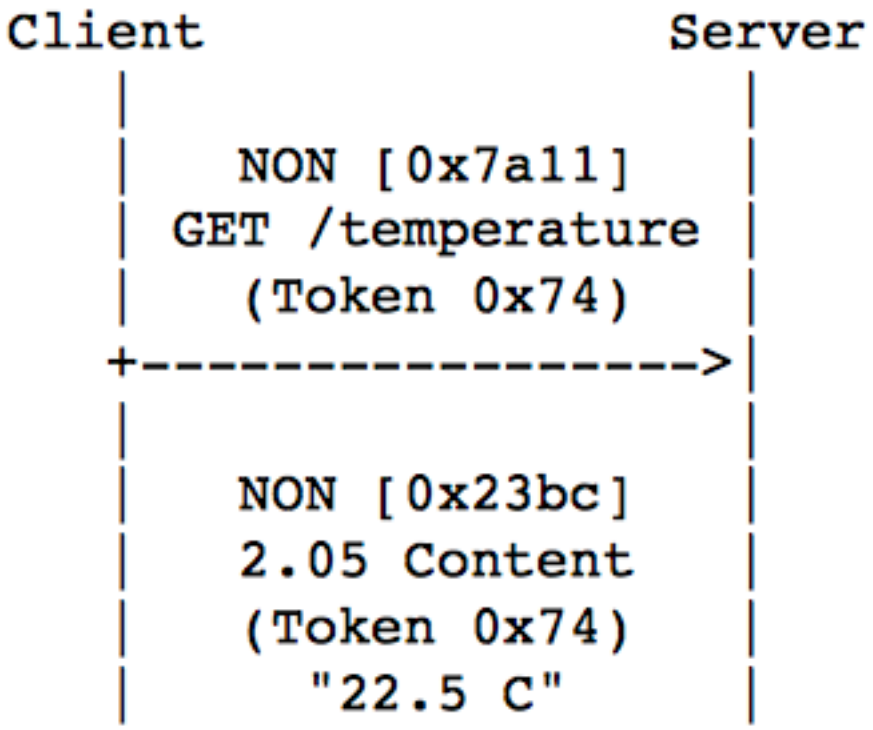




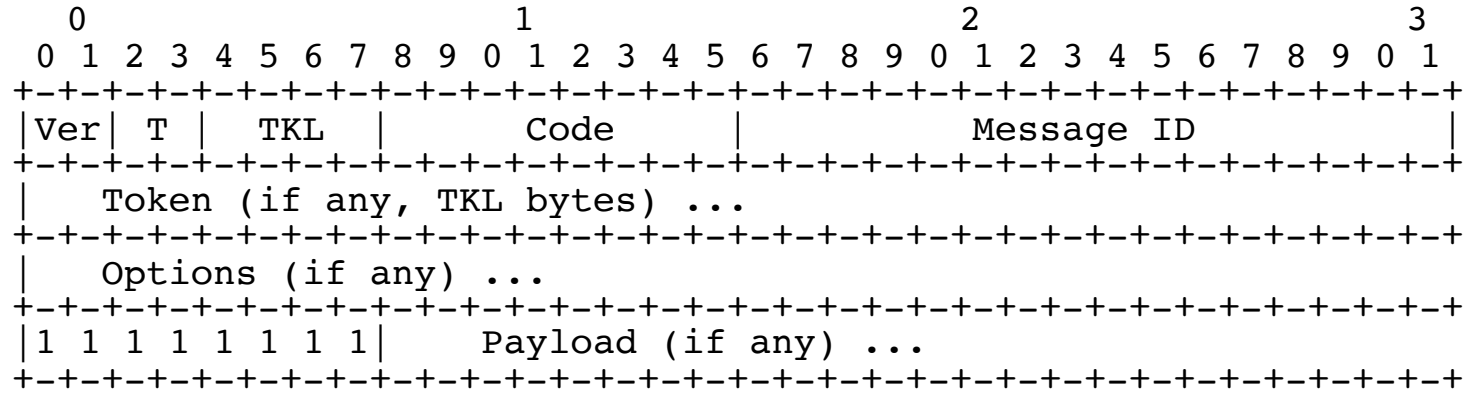
COAP: Separate Response



COAP: Non-confirmable Request



Message Header (4 bytes)



Ver - Version (1)

T – Message Type (Confirmable, Non-Confirmable, Acknowledgement, Reset)

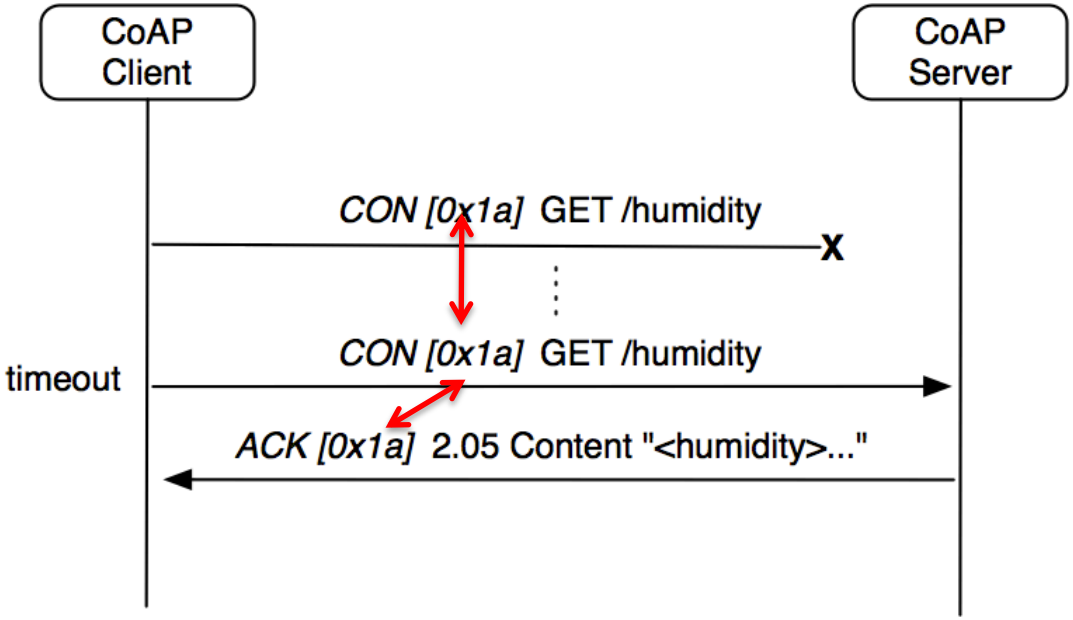
TKL- Token Length, if any, the number of Token bytes after this header

Code - Request Method (1-10) or Response Code (40-255)

Message ID – 16-bit identifier for matching responses

Token – Optional response matching token

Dealing with Packet Loss

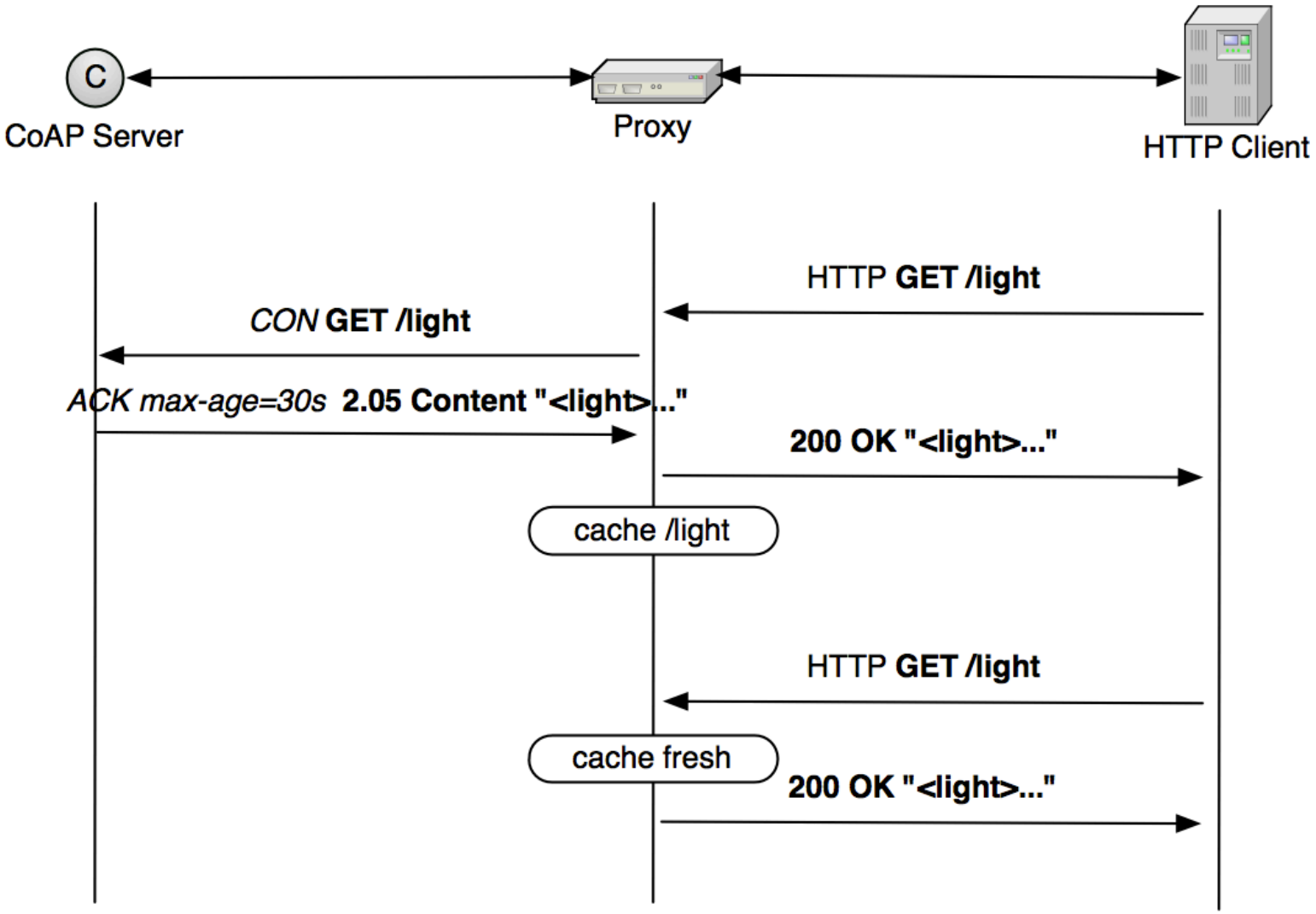


- Stop and Wait approach
- Repeat a request after a time-out in case ACK (or RST) is not coming back

Back-Off Details

- Initial time-out set to:
  - Rand [ACK_TIMEOUT, ACK_TIMEOUT * ACK_RANDOM_FACTOR] ([2s, 3s])
- When time-out expires and the transmission counter is less than MAX_RETRANSMIT (4)
  - retransmit
  - Increase transmission counter
  - double the time-out value
- The procedure is repeated until
  - A ACK is received
  - A RST message is received
  - the transmission counter exceeds MAX_RETRANSMIT
  - the total attempt duration exceeds MAX_TRANSMIT_WAIT (93s)

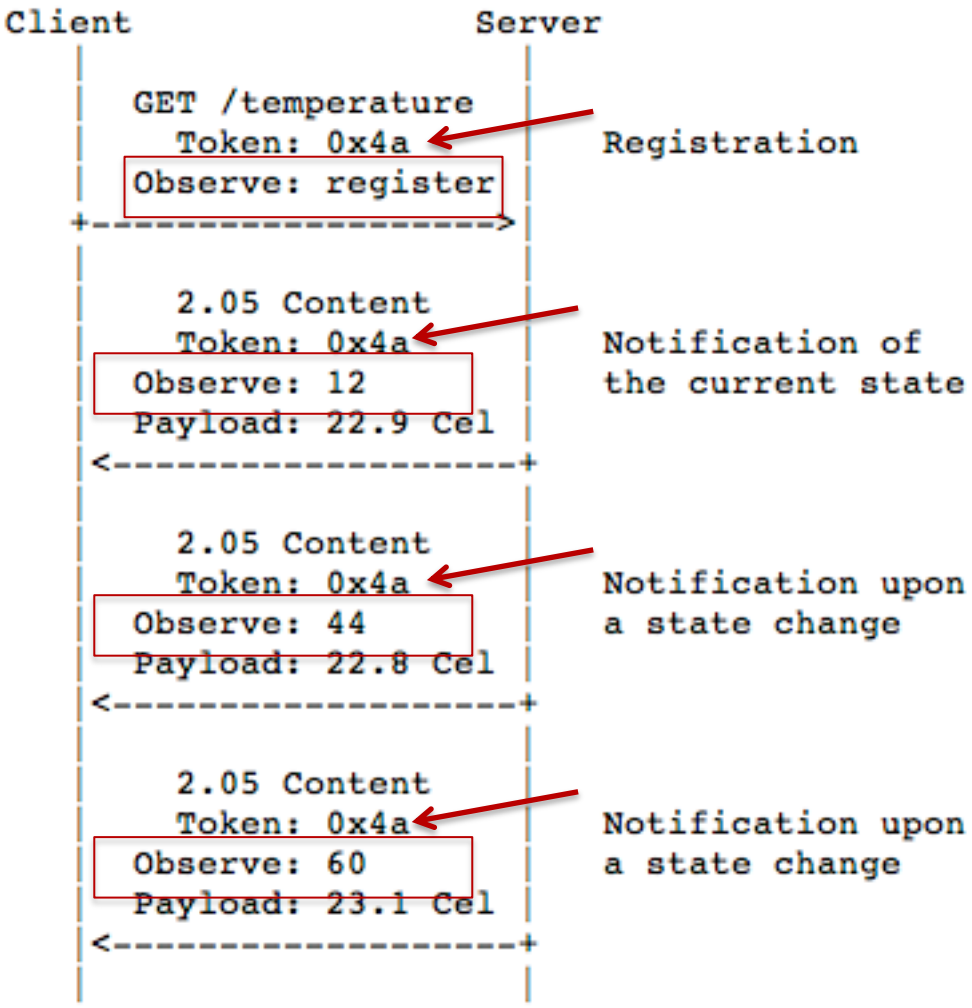
Proxyng and caching



COAP Observation

- PROBLEM:
  - REST paradigm is often “PULL” type, that is, data is obtained by issuing an explicit request
  - Information/data in WSN is often periodic/triggered (e.g., get me a temperature sample every 2 seconds or get me a warning if temperature goes below 5°C)
- SOLUTION: use Observation on COAP resources

Observation

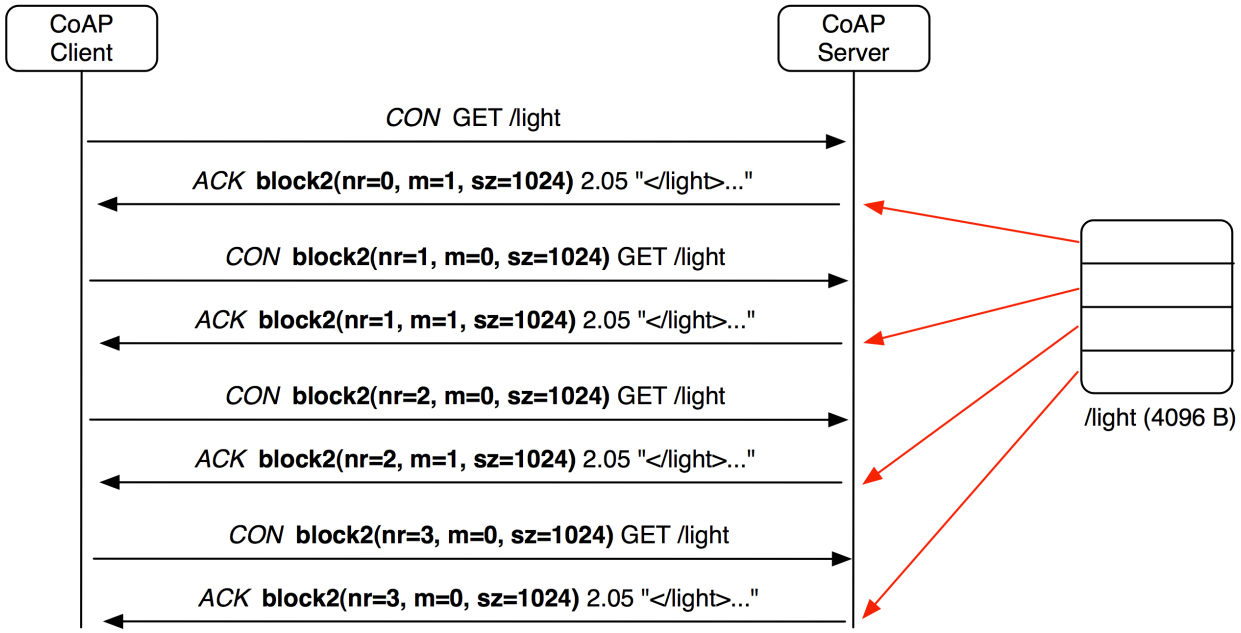


See draft-ietf-core-observe

COAP Block Transfer

- PROBLEM: avoid segmentation in the lower layers (IPv6)
- SOLUTION: COAP Block Transfer Mode
  - brings up fragmentation at the application layer

Block transfer



- *Block2* Option added to messages
  - nr=incremental block number within original data
  - m=more blocks flag
  - sz=block size

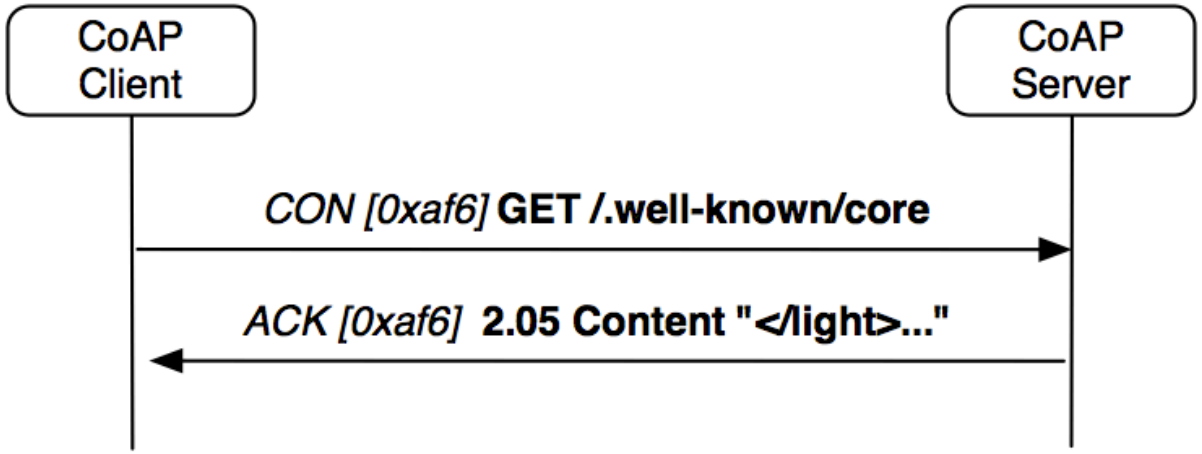
Discovery & Semantics

- Resource Discovery
  - GOAL: Discovering the links hosted by CoAP (or HTTP) servers

GET /.well-known/core?optional_query_string

  - Returns a link-header style format
    - URL, relation, type, interface, content-type etc.

CoRE Resource Discovery




```
</dev/bat>;obs;if="";rt="ipso:dev-bat";ct="0",
</dev/mdl>;if="";rt="ipso:dev-mdl";ct="0",
</dev/mfg>;if="";rt="ipso:dev-mfg";ct="0”,
</pwr/0/rel>;obs;if="";rt="ipso:pwr-rel";ct="0",
</pwr/0/w>;obs;if="";rt="ipso:pwr-w";ct="0",
</sen/temp>;obs;if="";rt="ucum:Cel";ct="0"
```

Getting Started with CoAP

- Open source implementations:
  - Java CoAP Library Californium
  - C CoAP Library Erbium
  - libCoAP C Library
  - jCoAP Java Library
  - OpenCoAP C Library
  - TinyOS and Contiki include CoAP support
- Firefox has a CoAP plugin called Copper
- Wireshark has CoAP plugin

The Message Queuing Telemetry Transport (MQTT)

Invented in 1999 (IBM proprietary standard)

Released in 2010

Official OASIS standard since 2014

Current version MQTT 3.1.1 http://docs.oasis-open.org/mqtt/mqtt/v3.1.1/mqtt-v3.1.1.html

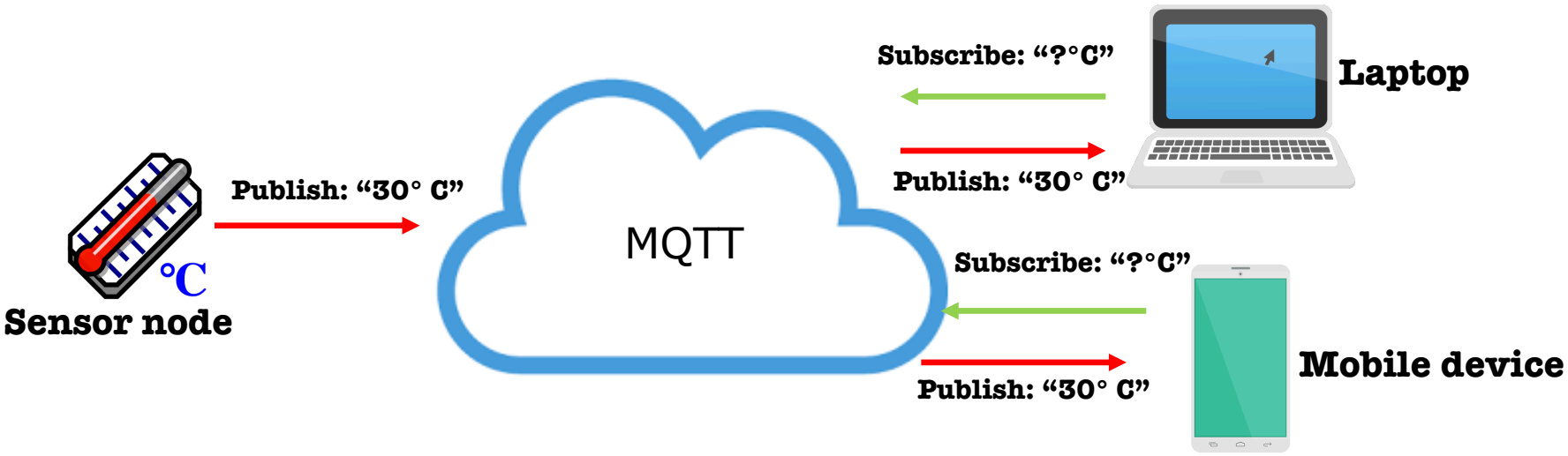
MQTT in short

MQTT is a Client Server publish/subscribe messaging transport protocol.

- More features:
  - Simple to implement (especially at the sensor side)
  - QoS Support
  - Lightweight and bandwidth efficient
  - Data agnostic
  - Session awareness

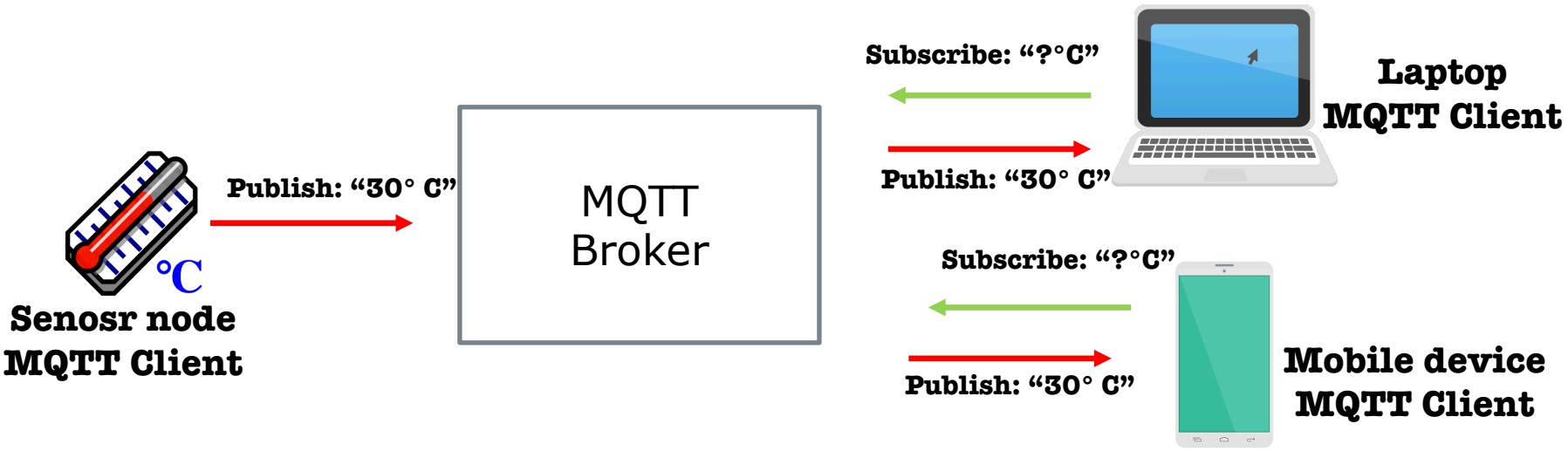
Communication Pattern

- Publish/Subscribe paradigm
  - Clients don't know each other
  - One-to-Many paradigm
  - Every client publishes & subscribes
  - PUSH information paradigm compared to PULL's one in COAP



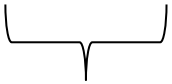
MQTT Components

- Publish/Subscribe paradigm
  - Clients don't know each other
  - One-to-Many paradigm
  - Every client publishes & subscribes
  - PUSH information paradigm compared to PULL's one in COAP



MQTT Topics

deib/antlab/room5/temperature



Topic level

Topic separator

- Wildcards allowed only when subscribing

deib/antlab/ + /temperature

Single level Wildcard

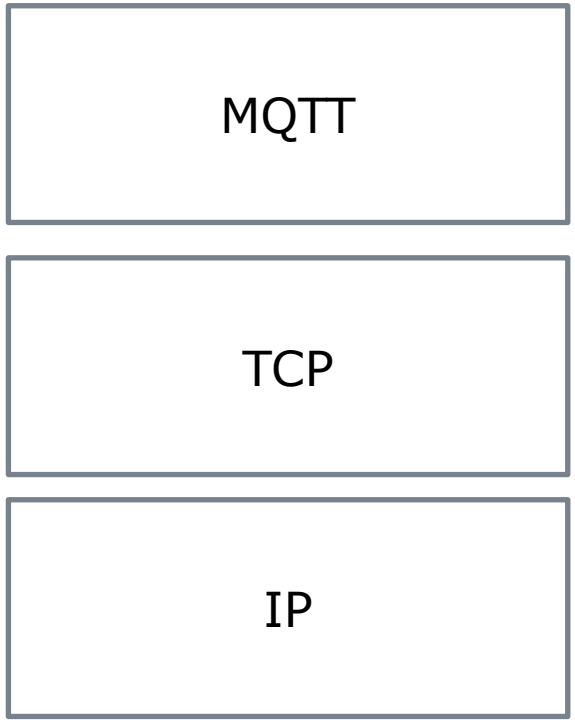
Plus sign can be used in multiple levels

Multi-level Wildcards

deib/antlab/room5/ #

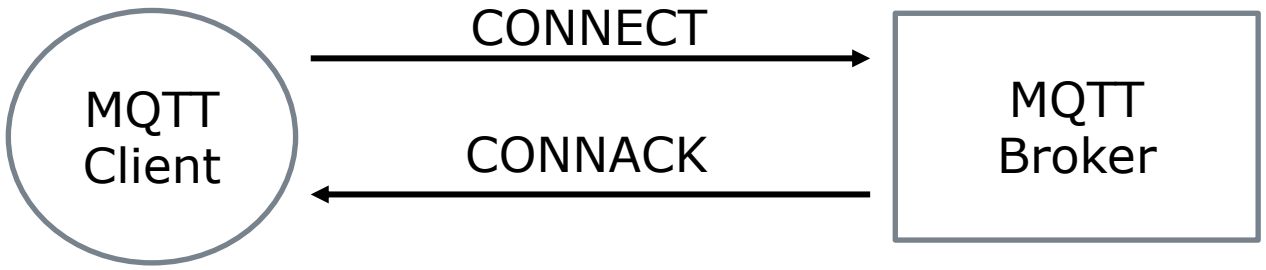
Multi-level Wildcard
(can be used only at the end)

MQTT Connections



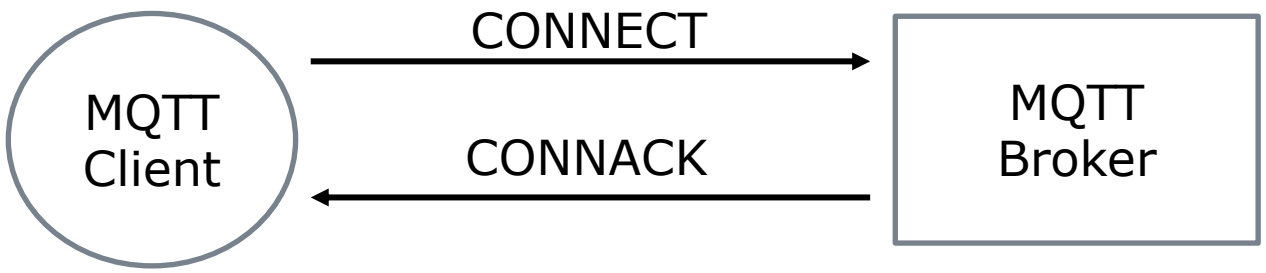
- Each MQTT client opens one connection to the MQTT Broker
- Push capabilities
- Works even through firewalls

Open Connections



- CONNECT message fields:
  - clientId “clientMatteo”
  - cleanSession true
  - username (opt) “matteo”
  - password (opt) “1234”
  - lastWillTopic (opt) “matteo/temp”
  - lastWillQoS (opt) 1
  - lastWillMessage (opt) “something wrong”
  - keepAlive 30

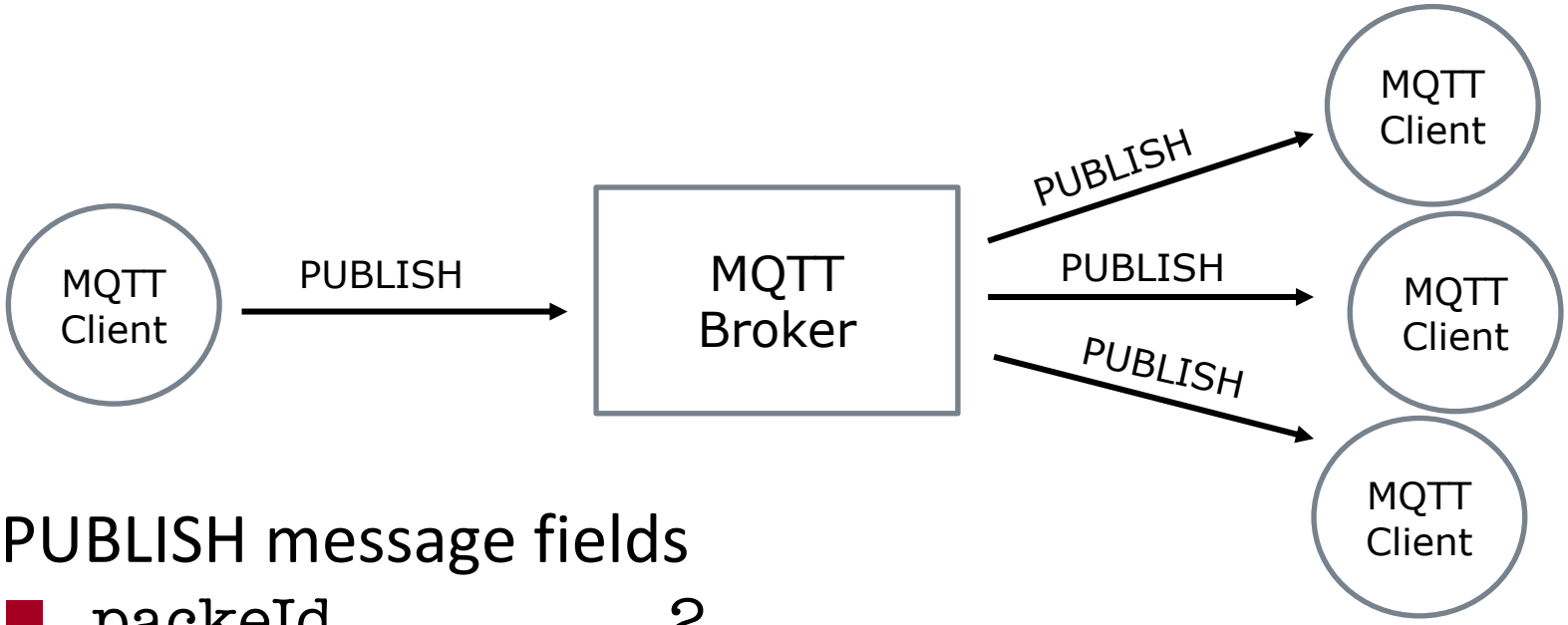
Open Connections



- CONNACK message fields:
  - sessionPresent true
  - returnCode 0-4

0: everything ok
1: unacceptable version
2: id rejected
3: server unavailable
4: bad username and pwd
5: unauthorized

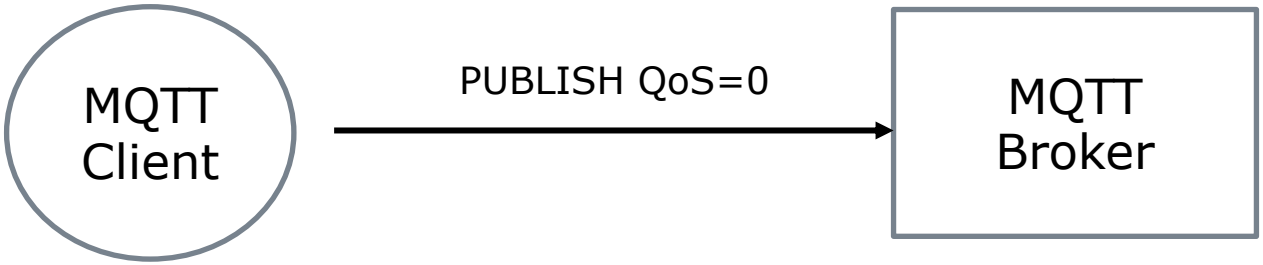
Publishing



- PUBLISH message fields
  - packeId 2
  - topicName "matteo/temp"
  - QoS 1
  - retainFlag false
  - Payload "temperature:30"
  - dupFlag false

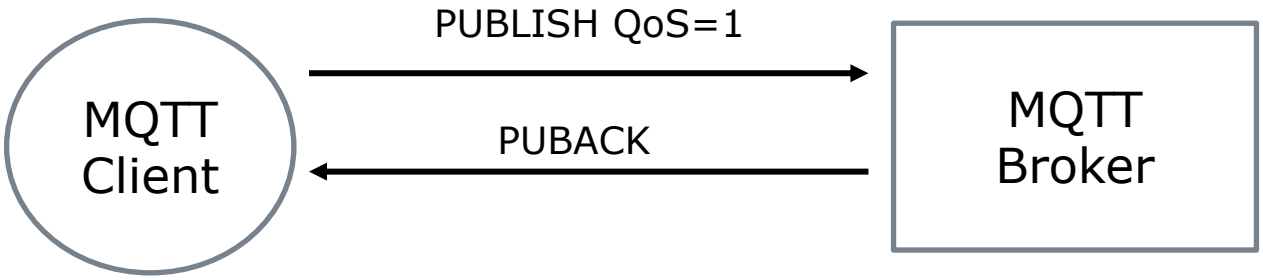
QoS 0: “at most once”

- Best effort transfer (same reliability provided by the underlying transport protocol)



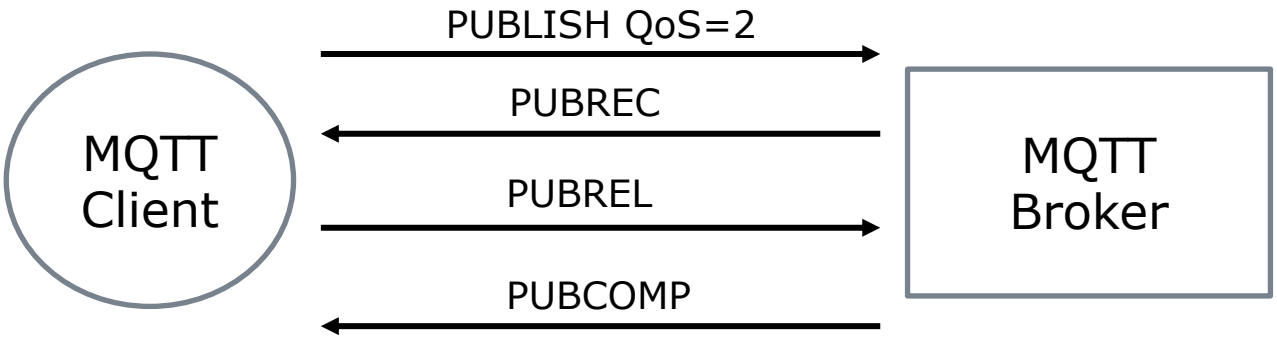
QoS 1: “at least once”

- The MQTT client stores the message and keeps retransmitting it until it is acknowledged by the MQTT broker (message can be received multiple times)



- PUBACK message fields:
  - `packetId` 2

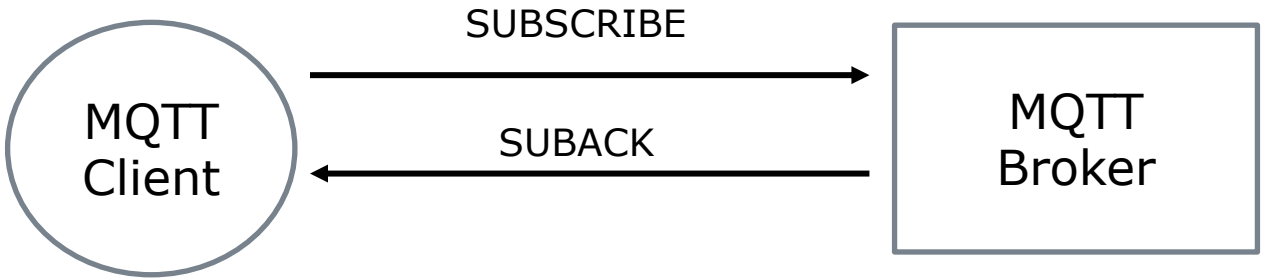
QoS 2: “exactly once”



- PUBREC, PUBREL, PUBCOMP message fields:
  - `PacketId 2`

- PUBLISH reception @ MQTT broker: process the packet accordingly, send PUBREC message back, store locally packetId to avoid duplicate processing
- PUBREC reception @ MQTT client: discard the initial packet and send PUBREL
- PUBREL reception @ MQTT broker: clear any current state and send PUBCOMP

Subscribing



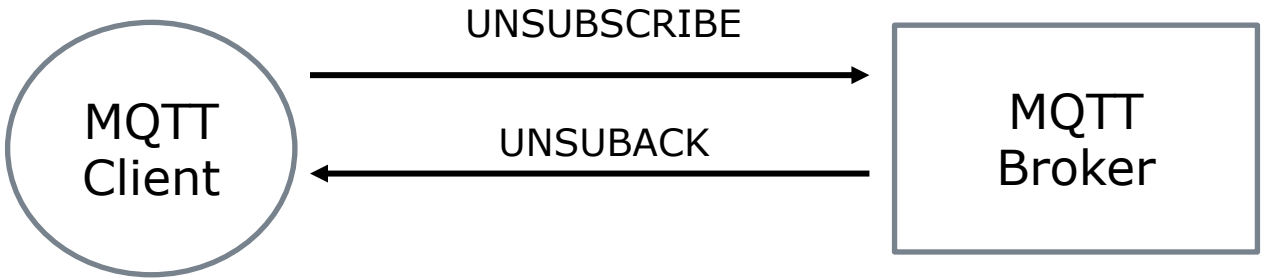
- SUBSCRIBE message fields:
  - packetId 2
  - QoS1 0
  - Topic1 “matteo/temp/1”
  - QoS2 1
  - Topic2 “kitchen/temp/2”
  - ....

one QoS/topic couple for each subscription

- SUBACK message fields:
  - packetId 2
  - returnCode 1 2
  - returnCode 2 0

one returnCode for each topic in the subscription

Unsubscribing



- UNSUBSCRIBE message fields:
  - packetId 2
  - Topic1 “matteo/temp/1”
  - Topic2 “kitchen/temp/2”
  - ....

- UNSUBACK message fields:
  - packetId 2

one returnCode for each topic in the subscription

Persistent Sessions

- In default operation mode when the client disconnects, all the client-related status at the broker is flushed (list of subscription, QoS pending messages, etc.)
- In persistent sessions both client and broker maintain a session:
  - Broker:
    - Existence of a session, even if there are no subscriptions
    - All subscriptions
    - All messages in QoS 1 or 2 flow, which are not confirmed by the client
    - All new QoS 1 or 2 messages, which the client missed while it was offline
    - All received QoS 2 messages, which are not yet confirmed to the client
  - Client
    - All messages in a QoS 1 or 2 flow, which are not confirmed by the broker
    - All received QoS 2 messages, which are not yet confirmed to the broker
- That means even if the client is offline all the above will be stored by the broker and are available right after the client reconnects.

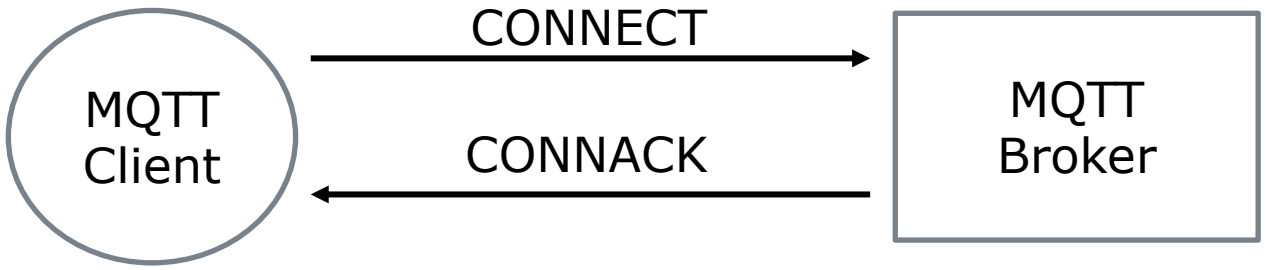
Retained Messages

- Problem: publishing and subscribing are asynchronous processes
- A client subscribing to a topic pattern may not get any message on that topic until some other client publishes on it
- Retained messages are PUBLISH messages with the retained Flag set to one
- The broker stores locally the retained message and send it to any other client which subscribes to a topic pattern matching that of the retained message

Last Will Message

- The Last Will and Testament (LWT) notifies other clients about an hard disconnect by a specific client
- Each client can specify its last will message when connecting to a broker
- The broker will store the message until it detects client hard disconnection
- The broker sends the message to all subscribed clients on the specific topic
- The stored LWT message will be discarded if a client disconnects gracefully by sending a DISCONNECT message.

LWT Set Up



- CONNECT message fields:
  - clientId “clientMatteo”
  - cleanSession true
  - username (opt) “matteo”
  - password (opt) “1234”
  - lastWillTopic (opt) “matteo/temp”
  - lastWillQoS (opt) 1
  - lastWillMessage (opt) “something wrong”
  - keepAlive 30

LWT Message is sent when..

- An I/O error or network failure is detected by the server.
- The client fails to communicate within the Keep Alive time.
- The client closes the network connection without sending a DISCONNECT packet first.
- The server closes the network connection because of a protocol error.

Keepalive

- It is responsibility of the client to keep the MQTT connection active
- Upon expiration of the keepalive, if no other interaction has happened with broker, the client ”pings” the broker which “pings it back”

- PINGREQ and PINGRESP messages have null payload

MQTT-SN

- Main differences
  - Extended architecture with Gateways and Forwarders
  - New gateway discovery procedures (and messages)
  - Some messages are more “compressed”
  - Extended keepalive procedures to support sleeping clients