

MQTT

© Riccardo Sisto, Politecnico di Torino

References for study:

Tutorials available on the internet (<https://mqtt.org/>)

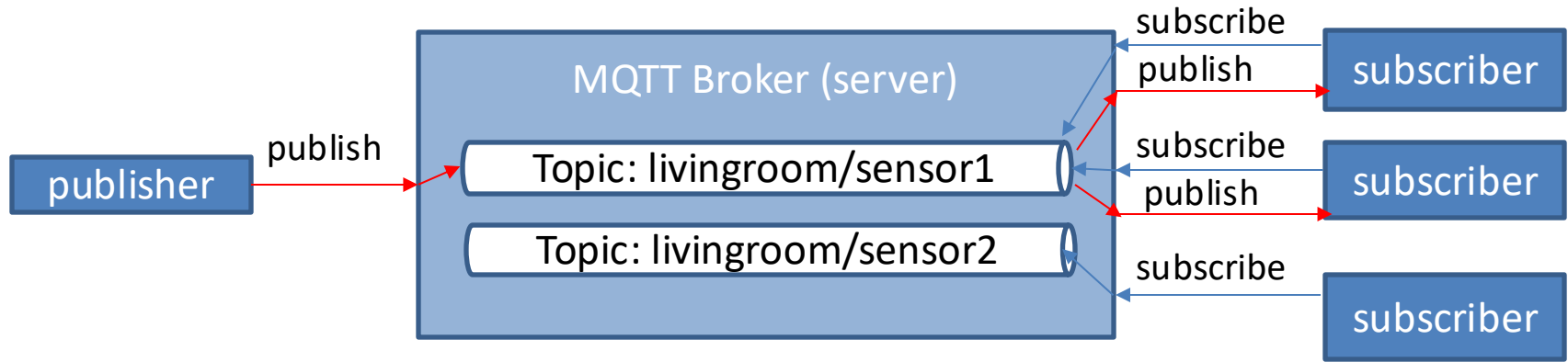
What is MQTT?

- MQTT: **MQ** Telemetry **T**ransport
- Client-server publish-subscribe messaging transport protocol, usually based on TCP
- Open, Lightweight, Simple
- Designed for constrained environments such as IoT M2M
 - need for small code footprint => **simplicity**
 - restricted network bandwidth, low-power => **efficiency**
 - unreliable communication channels (wireless, noisy, frequent disconnections) => **reliability and fault-tolerance**

MQTT History

- 1999 Designed as a proprietary protocol
 - IBM, satellite-connected Oil Pipeline Telemetry Systems
- 2010 Released royalty-free
- 2014 Submitted as OASIS Standard (MQTT 3.1)
- 2014 OASIS and ISO Standard (MQTT 3.1.1, 4 on the wire)
- 2019 new OASIS Standard (MQTT 5.0)
- MQTT-SN variant of MQTT for other transport protocols (e.g., ZigBee, UDP) Not yet a standard

MQTT Architecture



- Publish-Subscribe (different from Message Queue)
- Notifications normally delivered synchronously by Broker to connected subscribers
 - but it is possible to require retention/persistency mechanisms

MQTT Topics

- In publish-subscribe systems,
 - clients do not know each other (no referential coupling)
 - the addressable units are the channels (Topics in MQTT)
 - clients can communicate provided they know shared topics
- MQTT Topics are uniquely identified (within a broker) by hierarchical path-like names

can contain spaces

level separator

Name example: **apartment 1/livingroom/temperature**

- Topics are never created/destroyed
 - They exist per se, clients just need to reference them

Special Topics

- Topic names starting with \$ are reserved
 - cannot be used by applications for their own purposes
- Topic names starting with \$SYS provide system-related information
- Examples:
 - \$SYS/broker/version
 - \$SYS/broker/uptime
 - \$SYS/broker/clients/connected
 - \$SYS/broker/clients/disconnected
 - \$SYS/broker/clients/total

Topic Filters

- Topic filters can be used to refer to collections of topics by using wildcards:

single level

apartment 1/+ /temperature

multilevel
(must be at the end)

apartment 1/#

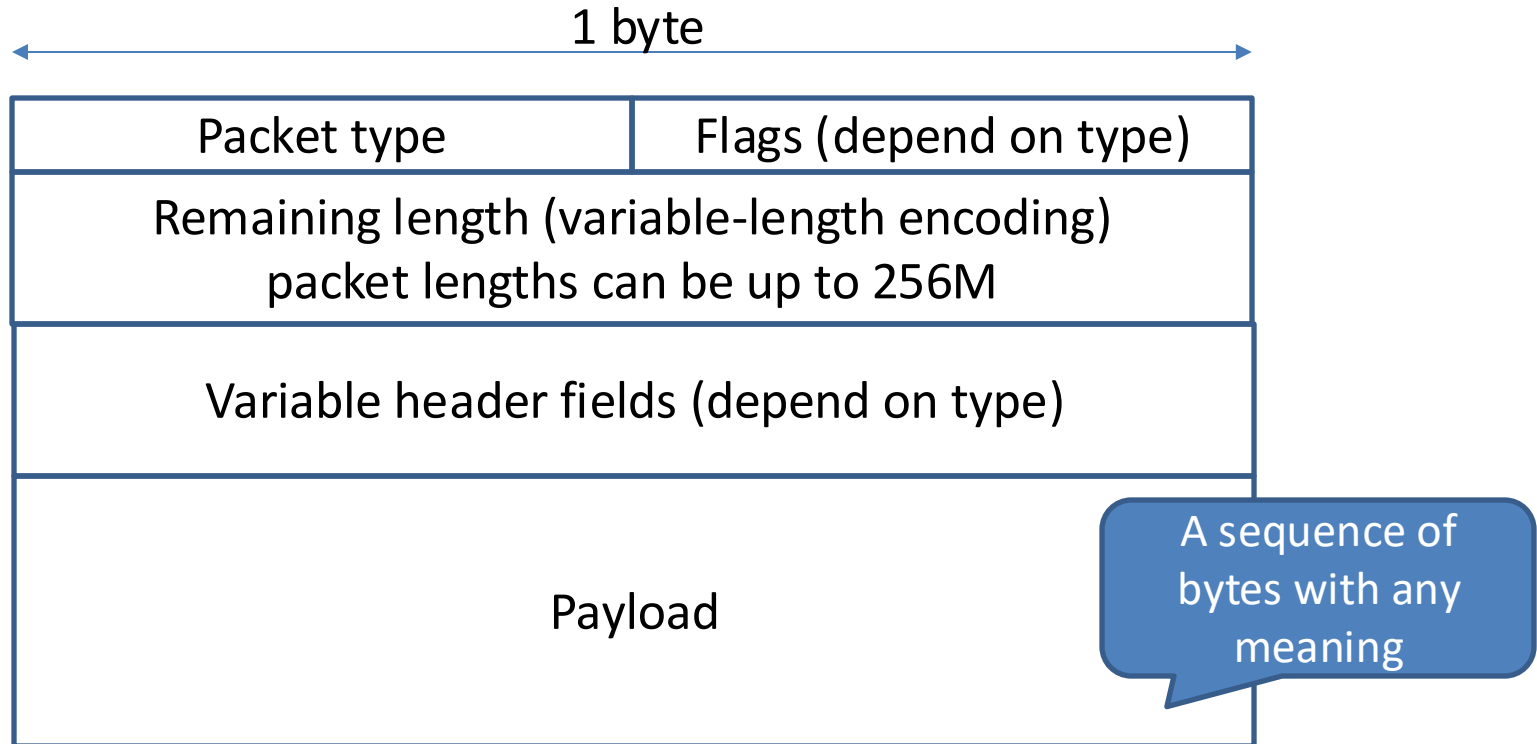
The MQTT Protocol

- Clients (publishers and subscribers) connect to the Broker via a layer-4 connection that provides a bidirectional ordered, lossless, stream of bytes (typically TCP or TCP+TLS)



- As TCP connections may fail, a client may be temporarily disconnected from the server

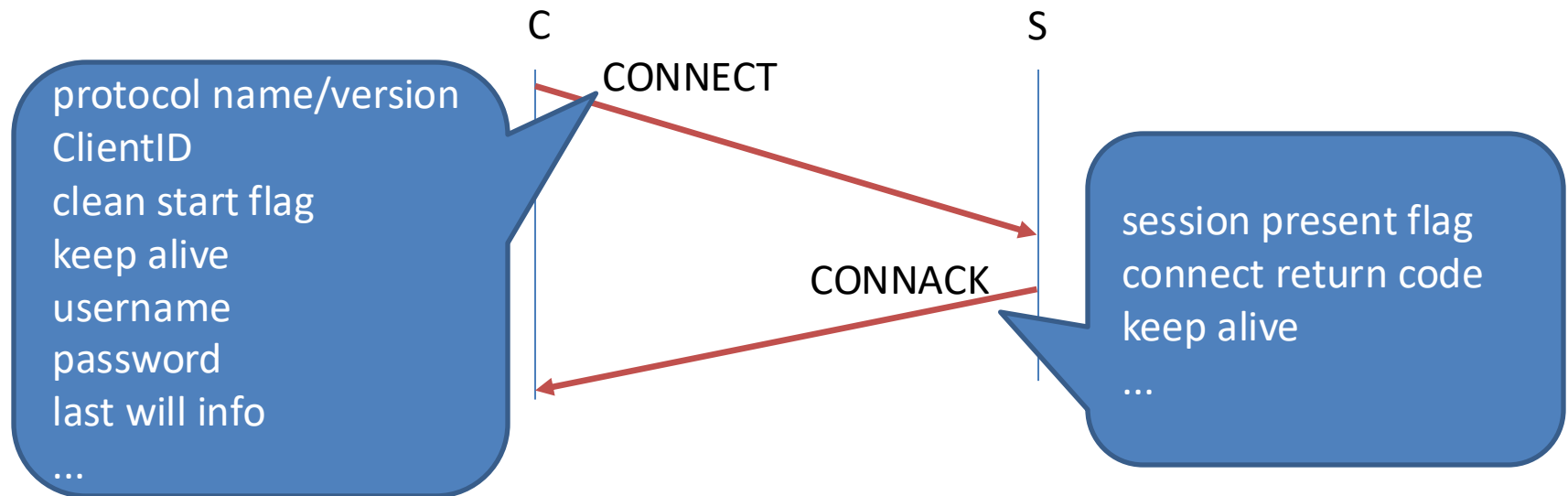
MQTT Control Packets



- Big-endian integers
- UTF-8 strings

The MQTT Handshake

- As soon as a connection is established, an MQTT request/response handshake takes place
 - client identification and authentication
 - association with client session state, if any
 - possibility to set/negotiate client preferences



Operations

- After the initial handshake, a client can perform a sequence of **operations** by exchanging control packets with the broker:
 - **Publish**
 - **Subscribe**
 - **Unsubscribe**
 - **Ping**
- A client terminates a session cleanly by sending a **Disconnect** control packet

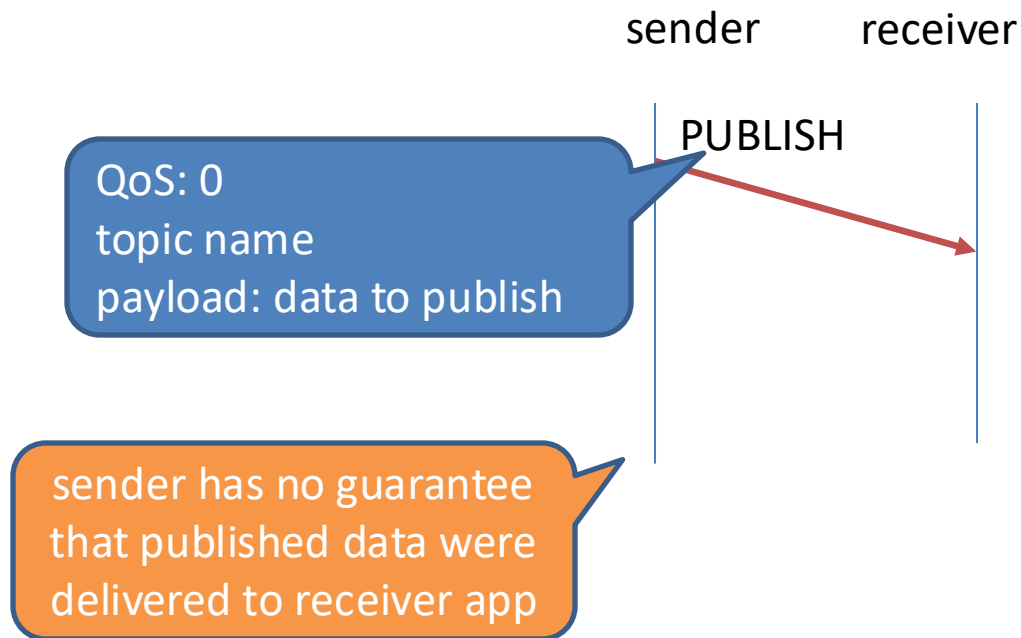
Publish and Quality of Service (QoS)

- A Publish operation starts with a publish request
 - topic
 - data to be published
- 3 possible QoS levels can be requested by Publish:
 - 0**: message delivered **at most once** (best effort)
 - 1**: message delivered **at least once** (may deliver multiple copies)
 - 2**: message delivered **exactly once**



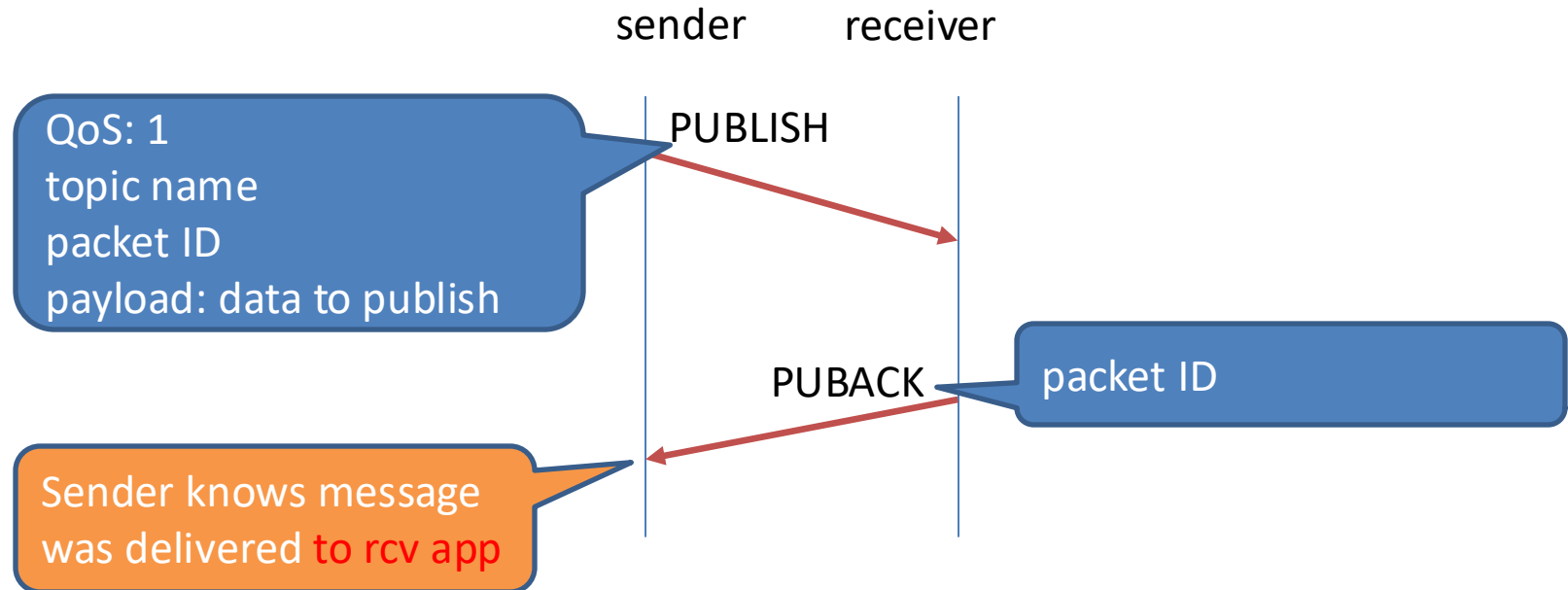
QoS 0 Publish Protocol

- At most once
 - fire and forget
 - nothing more than what is guaranteed by TCP



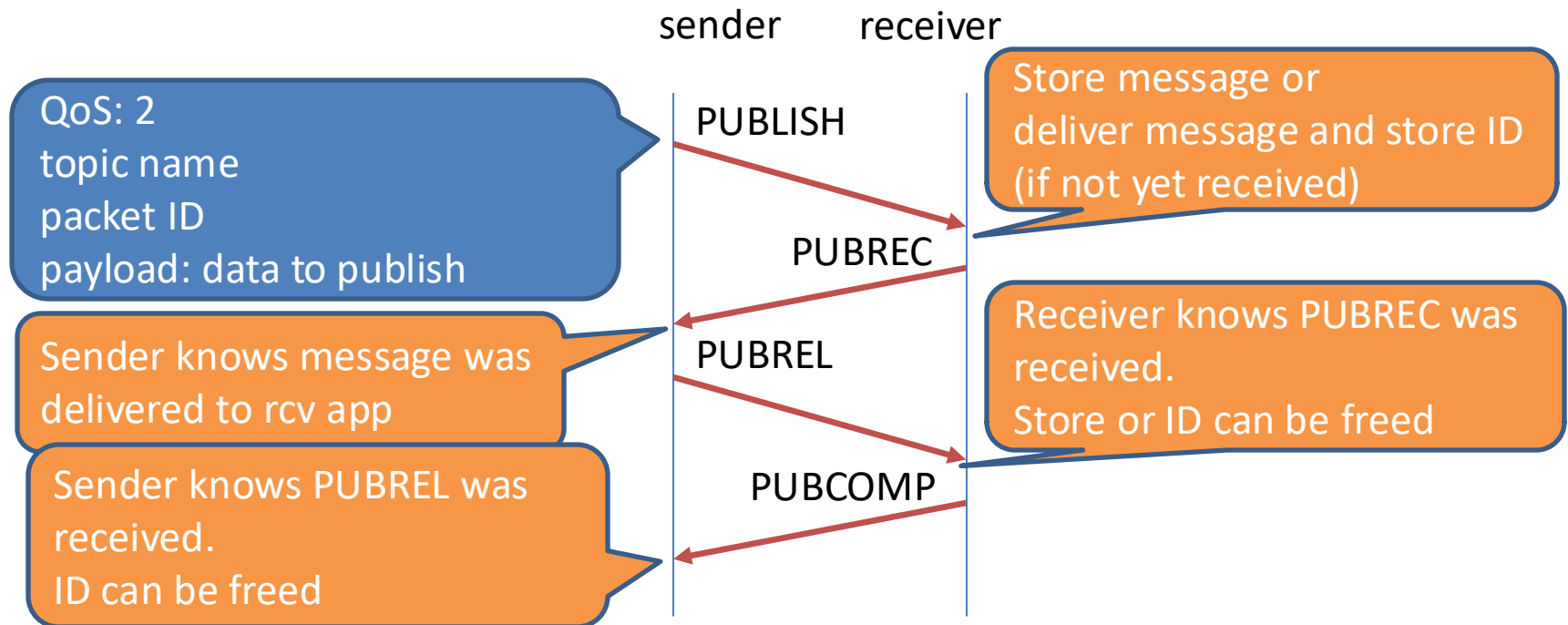
QoS 1 Publish Protocol

- At least once:
 - receiver acknowledges each delivery
 - sender waits for acknowledgment and retransmits after a timeout (sender forgets message only when it receives acknowledgment)



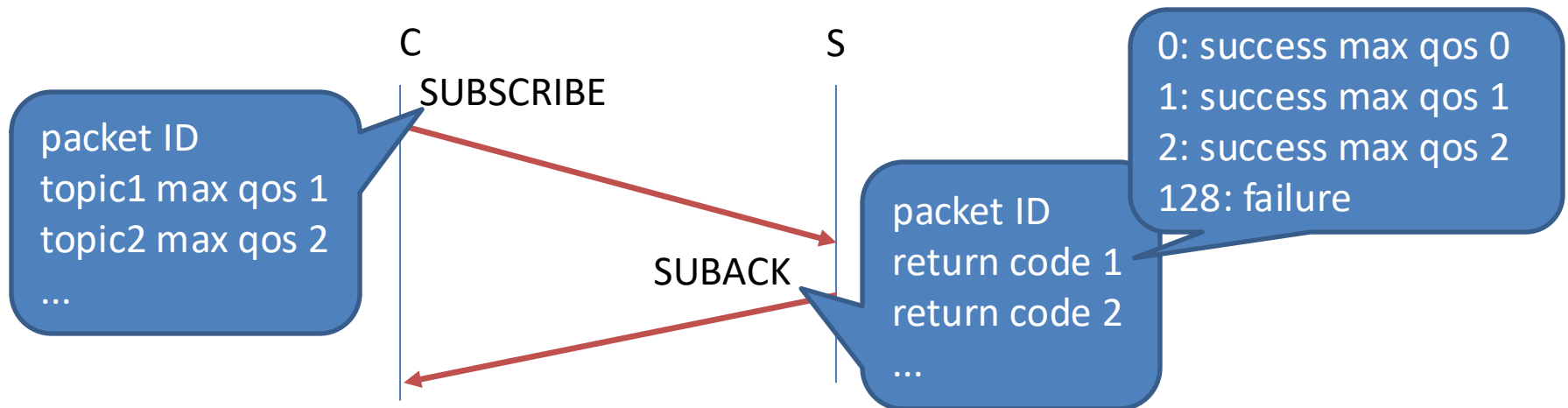
QoS 2 Publish Protocol

- Exactly Once:
 - in addition to delivery acknowledgment (as with QoS 1) a duplication avoidance mechanism is introduced:

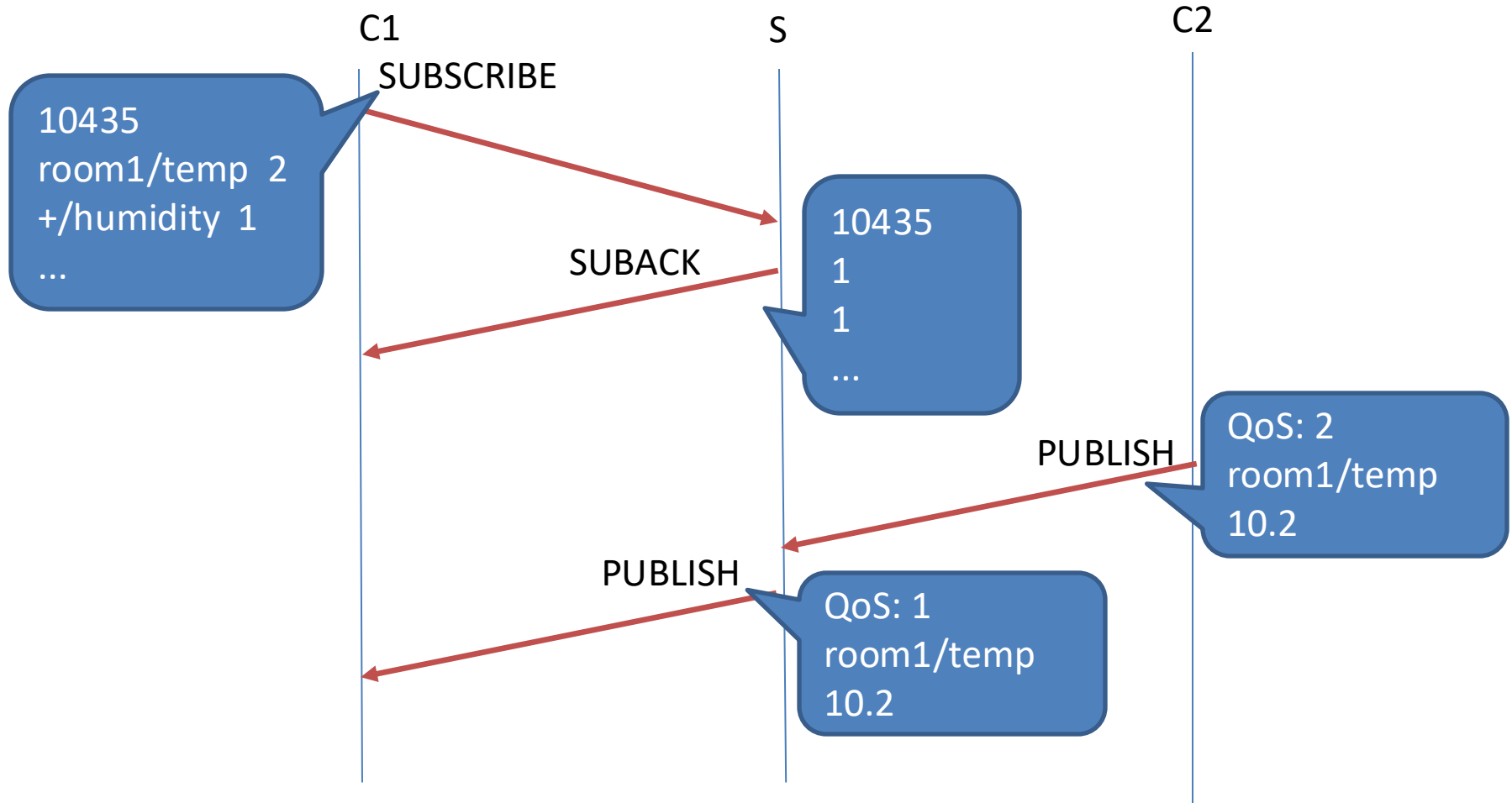


Subscription

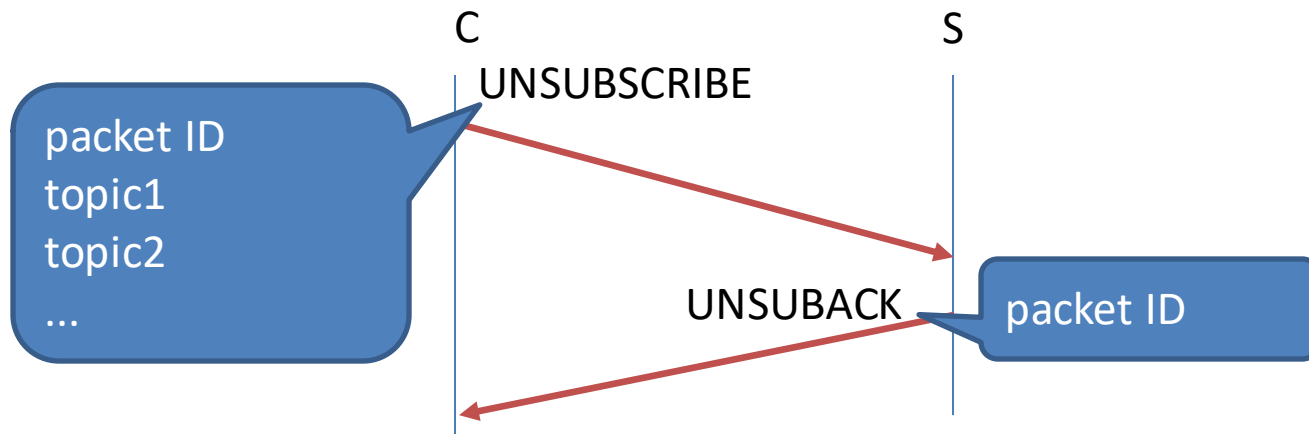
- One subscription can subscribe a client to one or more topics (expressed by one or more topic filters)
 - For each topic filter, a max QoS is requested
 - The server can grant subscriptions selectively, and downgrade the max QoS for each topic filter



Example: QoS Downgrading



Un-Subscription




Persistent Storage

- Broker maintains some state information even for disconnected subscribed clients
 - Persistent sessions (started with clean start flag unset)
 - subscriptions
 - In progress QoS 1 and QoS 2 publish
 - Possibility to mark a publish as RETAINED (boolean flag)
 - the last retained published value is kept by the broker for each topic
 - when a client subscribes to a topic with a retained value, that value is immediately published to the client
- Clients need to keep state information as well (in progress QoS 1 and QoS 2 publish)

Last Will (Testament)

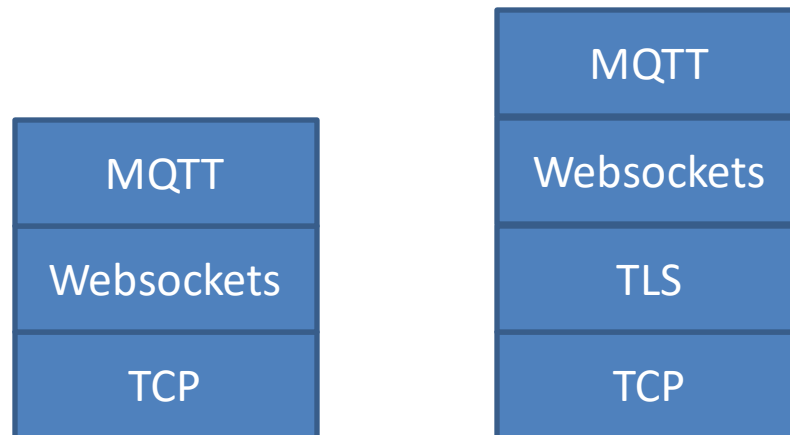
- Ungraceful disconnections of clients can be detected by the broker
- If the client that gets disconnected had a will, the broker will execute the will
 - i.e., publish the will message to the will topic
- A will can be set when connecting to the broker:
 - Will topic
 - Will message
 - Will QoS
 - Will retain flag



All fields are mandatory
when will is present

MQTT over Websockets

- Feature introduced to enable browsers to behave as MQTT clients
 - browser APIs do not offer direct access to TCP
 - but they offer access to websockets



Each MQTT packet is encapsulated into one or more websocket frames

MQTT Programming in Javascript

- Several MQTT client libraries available
- We experience the *MQTT-js* library
 - <https://www.npmjs.com/package/mqtt>
 - Available for both Node.js and the Browser
 - Can use MQTT over websockets

MQTT-js API

- Entry point: **mqtt**
 - Connection:
 - `mqtt.connect(url, options)` returns `mqtt.Client`
- Client connection: **mqtt.Client**
 - Events:

event name	meaning	callback parameters
connect	a connection has been opened	(connack)
close	a connection has been closed	()
message	a published message has been received	(topic, message, packet)
error	an error has occurred	(error)

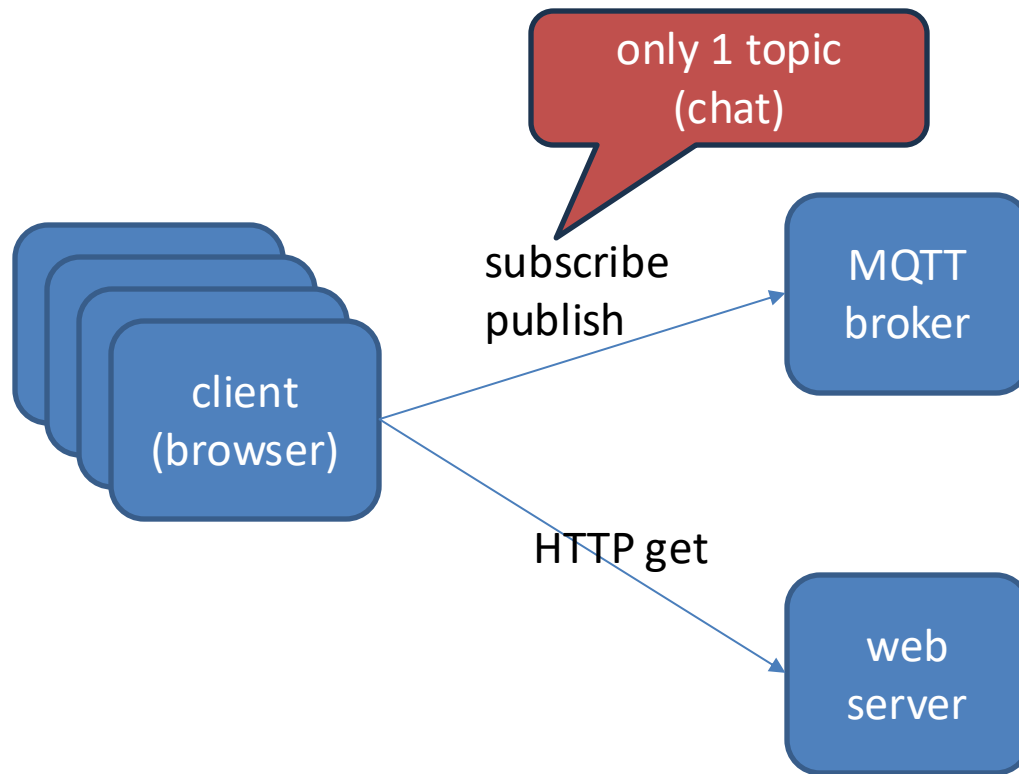
MQTT-js API

– Methods:

- `publish (topic, message, [options], [callback])`
- `subscribe (topic, [options], [callback])`
- `unsubscribe (topic, [options], [callback])`
- `end([force],[options], [callback])`

Example

- Re-implement the websockets chat using MQTT



Interface Design Approaches (Recall)

- **Method centric**
 - Fixed endpoint (classical programming language–like approach)
 - Design is about operations and their input/output arguments
- **Message centric**
 - Fixed, single-operation interface (e.g. `send(Message)`)
 - Design is about messages and endpoints
- **Constrained**
 - Fixed, multiple-operation interface
 - Example: REpresentational State Transfer (REST)
 - Design is about endpoints, allocation of operations, and their input/output arguments

Half way between the previous two ones

Pub/Sub Interface Design Guidelines

- Granularity Level of Messages and Topics
 - fine granularity of topics **reduces** the load on the broker
 - may depend on how interactive the application needs to be (but high interactivity has a cost)
- How to return errors
 - error topics and error messages
- Idempotent operations
 - relevant when messages may produce a state change
 - choice of topic semantics to have idempotent operations
- Documentation
 - message type + topic structure (better if self-explanatory)

Computing the Load on the Broker (and on the network)

- Frequency of publish operations: f
- Average number of topic subscribers: n
- Rate of messages (messages per second): $f(n+1)$
- Reducing topic granularity, n tends to decrease

MQTT Specific Guidelines

- Use plain ASCII in topic names
- Use topic names as identifiers to carry part of message information
- Leave topic structure open to extension