7. Application layer

Application layer protocols, data serialization

Wireless sensor networks

Martin Úbl
ublm@kiv.zcu.cz

2023/24

<DCSE> DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

Application layer

Basics

- application layer – ISO/OSI layer 7
- we usually group three sub-layers:
  - application sub-layer
  - presentation sub-layer
  - session sub-layer
- in this layer, we assume:
  - underlying protocols (L1 through L4) are implemented and performs their function
  - we don't care how the data gets delivered

Application layer

Basics

- *session sub-layer*
- rarely seen in WSN
- as it manages sessions, which are primarily a domain of point-to-point oriented approaches
- we will not discuss session sub-layer

Application layer

Basics

- *presentation sub-layer*
- manages data (de)serialization, compression and encryption
- we will discuss this
- data (de)serialization will be discussed in this presentation
- encryption and decryption will be discussed in future presentations
- compression will not be discussed separately
  - some serialization techniques considers compression by default

Application layer

Basics

- *application sub-layer*
- maintains data acquisition and transformation into streams
- maintains support roles in WSN and node operation
- we will discuss different aspects and also specific protocols

Application layer

Basics

- application layer in its entirety must consider a software architecture
- whole implementation tends to be monolithic
- tightly coupled components
- we will discuss a potential node software architecture (practicals)

Application layer

Protocols

- application layer protocols, e.g.:
  - data transfer
    - MQTT, or MQTT-SN
    - CoAP
  - node and network management
    - SMP
    - TADAP
    - SQDDP

Application layer

MQTT

- **Message Queuing Telemetry Transport** (MQTT)
- (MQ Telemetry Transport – a newer name)
- established protocol, even in the Internet
- *publish-subscribe* model
- *topics* for grouping data streams
- *subscribers* can *subscribe* to a *topic*
- *publishers* can *publish* to a *topic*
  - published message is sent to all *topic subscribers*
- a *broker* manages the communication
  - nodes subscribe and publish through it
  - it is a central node

Application layer

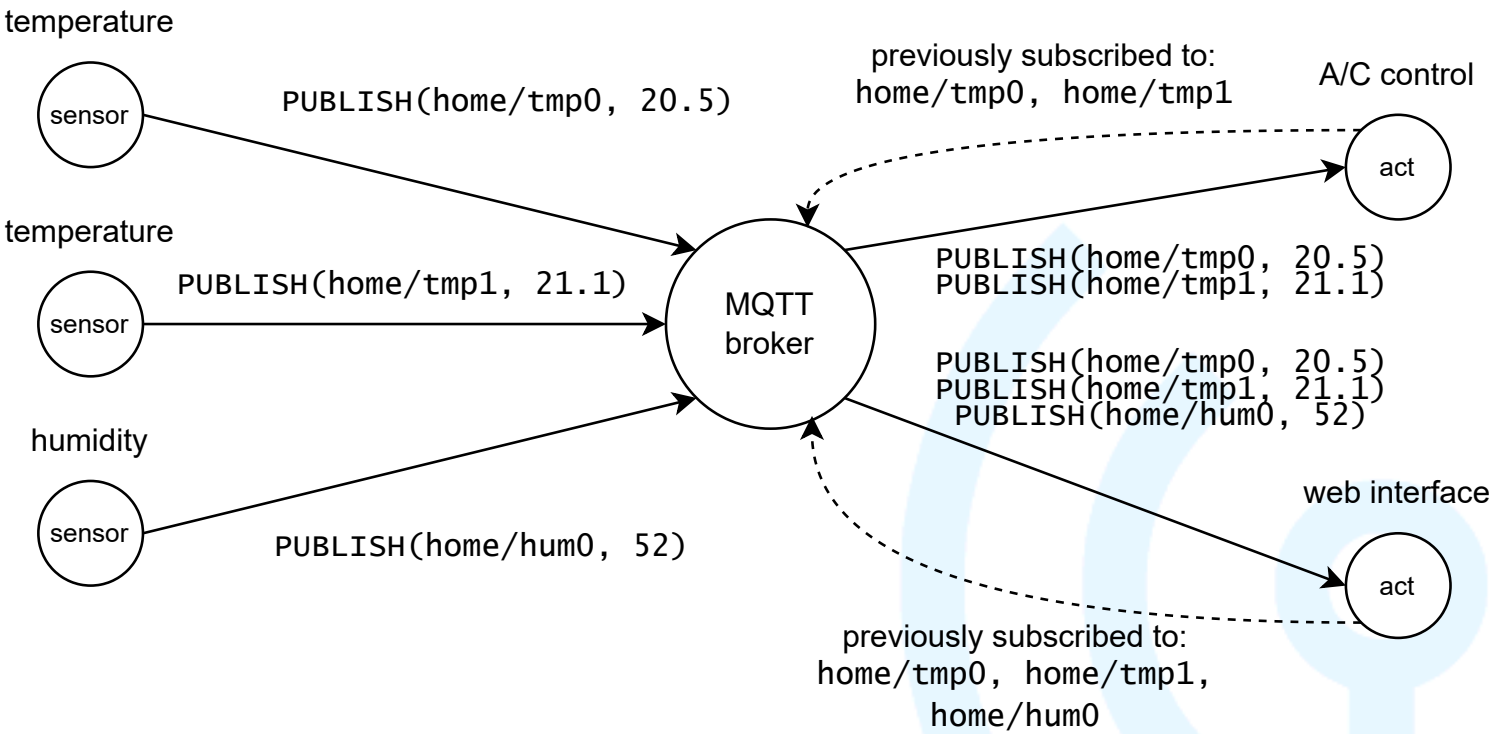
MQTT

- message types
  - CONNECT – a client connects to the server
  - CONNACK – the server acknowledges the client connection
  - PUBLISH – a client publishes to a topic; a server redistributes the message to all subscribed clients
  - PUBACK – the server acknowledges the published message
  - PUBREC, PUBREL, PUBCOMP – QoS level 2 assurement of message reception
  - SUBSCRIBE – a client subscribes to a topic
  - SUBACK – the server acknowledges the subscription
  - UNSUBSCRIBE – a client unsubscribes from a topic
  - UNSUBACK – the server acknowledges the unsubscription
  - PINGREQ, PINGRESP – keepalive mechanism
  - DISCONNECT – a client disconnects from the server

Application layer

MQTT

- architecture overview



Application layer

MQTT

- Quality of Services (QoS)
- defines 3 levels:
  - level 0 – "at-most-once"
    - nothing guarantees delivery; if the message arrives, it arrivest at most once
  - level 1 – "at-least-once"
    - guaranteed message delivery; may arrive multiple times
  - level 2 – "exactly-once"
    - guaranteed message delivery; arrives exactly once

Application layer

MQTT

- message payload
- MQTT is payload format agnostics
- for MQTT, it is just an array of bytes
- usually, we use strings and textual representation
  - very often combined with JSON, BSON or similar
- payload size is technically limited to 256 MB

Application layer

MQTT

- topic
- stream of data relevant to a single area of interest
- structured, uses / as delimiter – separates topic levels
- e.g., `home/bedroom/temperature`, `home/outside/motion`, ...
- supports wildcards
  - + for a single level wildcard
    - e.g., `/home/+/temperature` – all temperatures at home
  - # for a multi-level tail wildcard
    - e.g., `home/bedroom/#` – everything from bedroom

Application layer

MQTT

- implementations
- brokers (server-based)
  - Mosquitto
  - RabbitMQ
  - HiveMQ
  - ...
- clients (desktop apps)
  - Mosquitto (Client)
  - Paho (C/C++, Python, ...)
  - Async.MQTT5
  - MQTTnet (.NET-based)
  - MQTT.js (Javascript)
  - ...

Application layer

MQTT

- MQTT is not very suitable for WSN's
- too big overhead – can be as large as 12 bytes for a single message
- we would like to have a more compact protocol
- there is a sub-standard of MQTT for wireless sensor networks – MQTT-SN

Application layer

MQTT-SN

- **MQTT for Sensor Networks** (MQTT-SN)
- variant of MQTT for WSN
- reduces payload sizes to a bare minimum
- mostly compatible with MQTT
  - MQTT-SN commands can be mapped to MQTT commands
- *topic id* – 2 byte identificator of a topic
  - no need to transfer whole topic name each time
- *short topic name*
- *pre-defined topics*
- no need for permanent connection

Application layer

MQTT-SN

- MQTT gateway
  - a designated node, that supports both MQTT-SN and MQTT
  - translates MQTT-SN commands to MQTT commands and vice-versa
  - typically a role of a sink node or an edge node
- MQTT-SN supports gateway discovery protocol
- MQTT forwarder
  - merely forwards the MQTT-SN data
- supports node sleep

Application layer

MQTT-SN

- MQTT gateway operation modes
  - transparent gateway
    - one MQTT-SN link is mapped to a single MQTT connection
    - requires less constrained edge nodes
  - aggregating gatewayy
    - single MQTT connection for all MQTT-SN links

Application layer

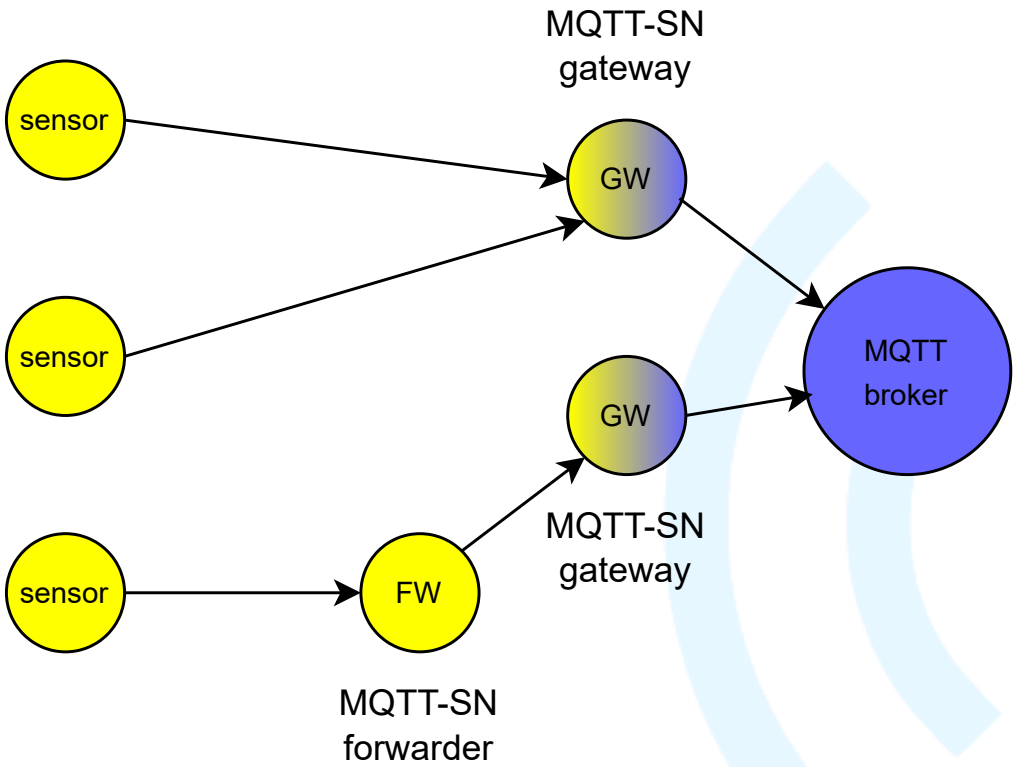
MQTT-SN

- topic name and identifiers
- *long topic name* – standard MQTT topic name
- *short topic name* – 2-byte identifier of the topic
- *pre-defined topic ID* – a 2-byte ID of previously registered topic
- topic registration
  - client sends a registration query with long topic name
  - broker/gateway responds with topic ID
  - from now on, client refer to this topic by its short ID

Application layer

MQTT-SN

- example of MQTT-SN configuration
  - yellow – MQTT-SN part
  - blue – MQTT part



Application layer

CoAP

- **Constrained Application Protocol** (CoAP)
- mainly for IoT applications, but can be used in WSN
- "lightweight" implementation of HTTP
- supports encryption via DTLS (L4 security)
- request-response, client-server model

Application layer

CoAP

- best way to describe the base functionality of CoAP is to describe it as *RESTful* API-like communication protocol
- it can be directly proxied to a HTTP server
- compact message structure
- has a binar header (4 bytes)
- optionally contains a payload
- supports GET, POST, PUT, DELETE
- supports URI
- supports subset of MIME types and HTTP response codes

Application layer

CoAP

- supports caching
- supports multicast
- optimizes delivery
- however... is not a "drop-in" replacement for HTTP
  - oversimplifies the communication
  - that is good for IoT and WSN, but not for a "large scale networks"

Application layer

SMP

- **Sensor Management Protocol** (SMP)
- protocol for managing sensor nodes
- sensor nodes usually don't have an unique address
  - or they do, but we don't want to use it, or we are unable to use it
- protocol for managing nodes based on their attributes
- can reconfigure the network
- e.g., change the location of the node
- e.g., manage retasking
- power cycles

Application layer

SMP

- SMP addresses nodes using their attributes or location
- not addressing a specific node
  - rather addressing a group of nodes that satisfy given criteria
- SMP not widely adapted
- however, attribute- and location-based addressing is adapted in almost all WSN management protocols
- SMP may perform e.g., key distribution, L2/L3/L4 reconfiguration, etc.

Application layer

SQDDP

- **Sensor Query and Data Dissemination Protocol** (SQDDP)
- data query protocol
- an example of a query:
  - *"Number of nodes, that detects light threshold over 30 %"*
  - *"Average temperature in the whole area"*
  - *"Maximum temperature of all sensors inside the house"*
- attribute- and location-based addressing
- supports aggregation
  - e.g., *maximum*, *average*, ...

Application layer

SQDDP

- query language: **Sensor Query and Tasking Language** (SQTL)
- running as a "service" on every node – can be seen as an implementation of SQDDP
- allows for defining events of three types:
  - *receive* – event is fired every time the node receives a message
  - *every* – event is fired periodicallyy
  - *expire* – event is fired once after a timer expires
- in fact, every implementation is individual, there is no strict rule set

Application layer

SQDDP

- usual operation types:
  - *sensor hardware access* – e.g., "get temperature", "turn on", "turn off"
  - *location-aware access* – e.g., "get neighbors", "get position"
  - *generic* – e.g., "tell", "execute"
- basically, the framework is generic enough to distribute code
  - *active networks*

Presentation layer

Data (de)serialization

- we have to discuss data (de)serialization
- once we have the data, how to transform them for transmission?
- what to consider:
  - payload size – the smaller the better
  - simplicity – simpler usually means less power spent on (de)serialization
  - data portability – data formats should be easily portable
  - code portability – the same code should run on both sides, even on servers, if required

Presentation layer

Data portability

- remark: data portability
- endianity – little vs. big endian
  - modern devices usually use little-endian
  - network endianity is usually big-endian
  - generic library must detect and convert, if required
- data type lengths
  - e.g., `long` data type – 8 bytes on LLP64, 4 bytes on most embedded devices
- support for data types
  - e.g., `float` is often supported, but `double` might not be
  - e.g., embedded devices often does not offer 8-byte data types at all

Presentation layer

Data portability

- `float` – floating point numbers
- it is generally better to avoid them
- use fixed-point numbers for transfers
- "emulate" fixed-point numbers by defining units
  - e.g., temperature of 21.8 ℃ → multiply by 10, so we might use integer data type and transfer the number 218

Presentation layer

Data portability

- `float`
- most of sensors have very limited precision with known domain
- there is no need for floating point
- e.g., a temperature sensor with range from -10 to 40 ℃ and precision up to 0.5 ℃
  - range: -10 to 40
  - precision: 0.5 (also interpreted the sampling period)
  - there is exactly $\frac{40-(-10)}{0.5} = 100$ possible values
    - $N = \frac{max-min}{precision}$
  - therefore, we may safely use a single byte unsigned integer type

Presentation layer

Data portability

- `float`
- transform to integer: $y_t = \frac{y - min}{precision}$
  - e.g., value of 10.5 is converted as $\frac{10.5-(-10)}{0.5} = 41$
- transform from integer: $y = y_t \cdot precision + min$
  - e.g., 41 back to original as $41 \cdot 0.5 + (-10) = 10.5$
- similar transformations are very common for WSN and embedded world

Presentation layer

Code portability

- remark: code portabilityy
- to avoid errors, code should be written once and used everywhere
- this is the reason why most of libraries for data (de)serialization is written in ANSI C
  - all desktop and server machines support ANSI C
  - all embedded devices support ANSI C
  - interoperability with ANSI C is very simple
    - on server-side, we may use Java and link ANSI C library via JNI

Presentation layer

Libraries

- **Protocolar Buffers** (protobuf)
- Google library for data (de)serialization
- has an embedded implementation: *nanopb*
- very convenient for WSN and IoT applications
- supports:
  - structures, flat and composite
  - lists, arrays, maps
  - enums, constants
  - implicit compression

Presentation layer

Protobuf

- not a library by its own
- more of a definition language with transpiler
- we define structures in `.proto` file, compile it to to obtain a code in our language
- we then use the generated code as a base

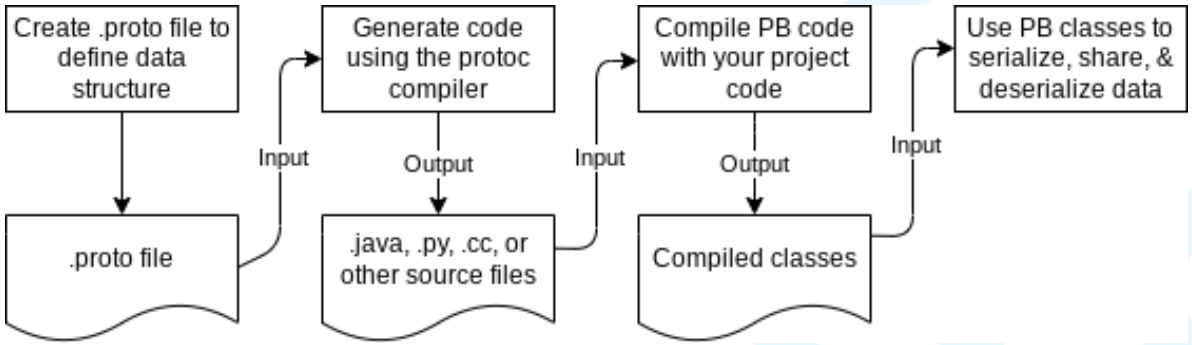


Figure: Protobuf scheme – taken from protobuf.dev

Presentation layer

Protobuf

- protobuf definition
- currently version 3 of the definition language
- defines data structures
- data types
  - `float`, `double`
    - ensures correct mapping to platform-dependent data types
  - `int32`, `int64`, `uint32`, `uint64`, `sint32`, `sint64`
    - uses variable-length encoding
    - variant with `s` prefix better encodes negative values
  - `fixed32`, `fixed64`, `sfixed32`, `sfixed64`
    - uses fixed-length encoding
  - `bool`
  - `string`
  - `bytes`

Presentation layer

Protobuf

- additional markings:
  - `optional` – the field is not mandatory
  - `repeated` – there ma be more than one instance, just like an array/vector of values
  - `map` – an associative container of key-value pairs

Presentation layer

Protobuf

- Example `person.proto` file

```
syntax = "proto3";

package PERSON;

message PersonInfo {
   string name = 1;
   int32 id = 2;
   optional string email = 3;
};
```

Presentation layer

Protobuf

- Translate to C++

```
protoc -I=./ --cpp_out=./ person.proto
```

- this generates two files:
  - `person.pb.cc` – implementation of (de)serialization of person structure
  - `person.pb.h` – interface file with generated structure definition

Presentation layer

Protobuf

- We can now use generated structures in out C++ code

```
PERSON::Person ceoPerson;

ceoPerson.set_name("Big Boss");
ceoPerson.set_id(1);
ceoPerson.set_email("boss@company.com");

size_t size = ceoPerson.ByteSizeLong();
std::vector<uint8_t> buffer(size);
address_book.SerializeToArray(buffer.data(),
                              size);
```

- the `buffer` vector now holds the serialized person

Presentation layer

Protobuf

- more on practicals
- protocolar buffers are established way to build presentation layer
- for embedded side: *nanopb* or similar
  - minimal memory footprint
  - fast operation
  - may be limited
  - works like the original protobuf – generates C code from the same `.proto` definitions
  - generated code must be compatible

Presentation layer

Serialization

- other formats and techniques
  - *Flatbuffers* – modern way, compatible with large scale of languages
  - *PSON* – effective binary encoding format
  - *MessagePack* – very compact format, but also very limited (data types)
  - etc.

Presentation layer

Serialization

- there are obvious reasons, why not use the "big world" encodings
  - JSON, XML, YAML, ...
- JSON fits well dynamically typed languages, but are 4-10 times more verbose
- XML is extremely verbose, it benefits from clearly defined structure
- YAML relies on indentation for multi-level definitions – this means additional overhead characters
- we surely want to use binary encoding, as it is much more compact and saves much traffic
- smaller messages $\rightarrow$ lower probability of error during transmission $\rightarrow$ less energy used

Presentation layer

Encryption

- presentation sub-layer maintains encryption and decryption
- more on security – next lectures
- large scale of security issues in terms of WSN
- we will also discuss some targetted attacks