International Telecommunication Union

ITU-T

TELECOMMUNICATION
STANDARDIZATION SECTOR
OF ITU

Q.3055
(12/2019)

SERIES Q: SWITCHING AND SIGNALLING, AND ASSOCIATED MEASUREMENTS AND TESTS

Signalling requirements and protocols for the NGN – Network signalling and control functional architecture

Signalling protocol for heterogeneous Internet of things gateways

Recommendation ITU-T Q.3055

ITU-T Q-SERIES RECOMMENDATIONS

SWITCHING AND SIGNALLING, AND ASSOCIATED MEASUREMENTS AND TESTS

SIGNALLING IN THE INTERNATIONAL MANUAL SERVICE	Q.1–Q.3
INTERNATIONAL AUTOMATIC AND SEMI-AUTOMATIC WORKING	Q.4–Q.59
FUNCTIONS AND INFORMATION FLOWS FOR SERVICES IN THE ISDN	Q.60–Q.99
CLAUSES APPLICABLE TO ITU-T STANDARD SYSTEMS	Q.100–Q.119
SPECIFICATIONS OF SIGNALLING SYSTEMS No. 4, 5, 6, R1 AND R2	Q.120–Q.499
DIGITAL EXCHANGES	Q.500–Q.599
INTERWORKING OF SIGNALLING SYSTEMS	Q.600–Q.699
SPECIFICATIONS OF SIGNALLING SYSTEM No. 7	Q.700–Q.799
Q3 INTERFACE	Q.800–Q.849
DIGITAL SUBSCRIBER SIGNALLING SYSTEM No. 1	Q.850–Q.999
PUBLIC LAND MOBILE NETWORK	Q.1000–Q.1099
INTERWORKING WITH SATELLITE MOBILE SYSTEMS	Q.1100–Q.1199
INTELLIGENT NETWORK	Q.1200–Q.1699
SIGNALLING REQUIREMENTS AND PROTOCOLS FOR IMT-2000	Q.1700–Q.1799
SPECIFICATIONS OF SIGNALLING RELATED TO BEARER INDEPENDENT CALL CONTROL (BICC)	Q.1900–Q.1999
BROADBAND ISDN	Q.2000–Q.2999
SIGNALLING REQUIREMENTS AND PROTOCOLS FOR THE NGN	Q.3000–Q.3709
General	Q.3000–Q.3029
Network signalling and control functional architecture	**Q.3030–Q.3099**
Network data organization within the NGN	Q.3100–Q.3129
Bearer control signalling	Q.3130–Q.3179
Signalling and control requirements and protocols to support attachment in NGN environments	Q.3200–Q.3249
Resource control protocols	Q.3300–Q.3369
Service and session control protocols	Q.3400–Q.3499
Service and session control protocols – supplementary services	Q.3600–Q.3616
Service and session control protocols – supplementary services based on SIP-IMS	Q.3617–Q.3639
VoLTE/ViLTE network signalling	Q.3640–Q.3655
NGN applications	Q.3700–Q.3709
SIGNALLING REQUIREMENTS AND PROTOCOLS FOR SDN	Q.3710–Q.3899
TESTING SPECIFICATIONS	Q.3900–Q.4099
SIGNALLING REQUIREMENTS AND PROTOCOLS FOR IMT-2020	Q.5000–Q.5049
COMBATING COUNTERFEITING AND STOLEN ICT DEVICES	Q.5050–Q.5069

For further details, please refer to the list of ITU-T Recommendations.

Recommendation ITU-T Q.3055

Signalling protocol for heterogeneous Internet of things gateways

Summary

A heterogeneous gateway is a hardware and software system used for the interaction of Internet of things (IoT) devices with each other and with remote IoT services. It comprises an IoT gateway infrastructure (IoT GI) and a semantic IoT gateway (SIoTG). The IoT GI is a hardware and software system comprising IoT device hardware, network interfaces, operating system, and simulator or virtualization system (virtual machine, containers, etc.). It is used to ensure compatibility of the different network technologies. The SIoTG is a software program and part of the heterogeneous IoT gateway, and is used for the mapping of different protocols, applications and IoT services among themselves.

The SIoTG that forms part of the heterogeneous gateway is supposed to ensure the mapping of different IoT solutions (protocols, applications and services) among themselves, independently of the configuration of the IoT devices themselves. Use of scenarios for the mapping of IoT solutions among themselves without using a special signalling protocol for the SIoTG is possible only if a standard configuration is used for each solution (e.g., for the mapping of protocols: a standard network port number for the protocol being mapped; no encryption of data embedded in the message, etc.). Otherwise, there is no standard solution for mapping procedures, and the need then arises for interaction with the SIoTG for the purpose of configuring the IoT solution mapping scenario.

Recommendation ITU-T Q.3055 describes the signalling protocol for heterogeneous IoT gateways.

History

Edition	Recommendation	Approval	Study Group	Unique ID[*]
1.0	ITU-T Q.3055	2019-12-14	11	11.1002/1000/14141

Keywords

Gateway, heterogeneous, Internet of things, protocols, semantic.

[*] To access the Recommendation, type the URL http://handle.itu.int/ in the address field of your web browser, followed by the Recommendation's unique ID. For example, http://handle.itu.int/11.1002/1000/11830-en.

Table of Contents

Recommendation ITU-T Q.3055

Signalling protocol for heterogeneous Internet of things gateways

1 Scope

This Recommendation describes the signalling protocol for heterogeneous Internet of things gateways. In particular, it:

- Addresses the overall network model necessary for mapping and signalling procedures;
- Addresses the software architecture necessary for mapping and signalling procedures;
- Describes possible scenarios for the use of signalling.

2 References

The following ITU-T Recommendations and other references contain provisions which, through reference in this text, constitute provisions of this Recommendation. At the time of publication, the editions indicated were valid. All Recommendations and other references are subject to revision; users of this Recommendation are therefore encouraged to investigate the possibility of applying the most recent edition of the Recommendations and other references listed below. A list of the currently valid ITU-T Recommendations is regularly published. The reference to a document within this Recommendation does not give it, as a stand-alone document, the status of a Recommendation.

[ITU-T Q.4060]	Recommendation ITU-T Q.4060 (2018), *The structure of the testing of heterogeneous Internet of things gateways in a laboratory environment*.
[ITU-T Y.4000]	Recommendation ITU-T Y.4000/Y.2060 (2012), *Overview of the Internet of things*.
[ITU-T Y.4050]	Recommendation ITU-T Y.4050/Y.2069 (2012), *Terms and definitions for the Internet of things*.
[ITU-T Y.4100]	Recommendation ITU-T Y.4100/Y.2066 (2014), *Common requirements of the Internet of things*.
[ITU-T Y.4101]	Recommendation ITU-T Y.4101/Y.2067 (2017), *Common requirements and capabilities of a gateway for Internet of things applications*.
[ITU-T Y.4113]	Recommendation ITU-T Y.4113 (2016), *Requirements of the network for the Internet of things*.
[ITU-T Y.4418]	Recommendation ITU-T Y.4418 (2018), *Gateway functional architecture for Internet of things applications*.
[IETF RFC 4122]	IETF RFC 4122 (2005), *A Universally Unique IDentifier (UUID) URN Namespace*.
[IETF RFC 7574]	IETF RFC 7574 (2015), *Peer-to-Peer Streaming Peer Protocol (PPSPP)*.

3 Definitions

3.1 Terms defined elsewhere

This Recommendation uses the following terms defined elsewhere:

3.1.1 device [ITU-T Y.4000]: With regard to the Internet of things, this is a piece of equipment with the mandatory capabilities of communication and the optional capabilities of sensing, actuation, data capture, data storage and data processing.

3.1.2 Internet of things (**IoT**) [ITU-T Y.4000]: A global infrastructure for the information society, enabling advanced services by interconnecting (physical and virtual) things based on existing and evolving interoperable information and communication technologies.

3.2 Terms defined in this Recommendation

This Recommendation defines the following terms:

3.2.1 data aggregator: Either end-point data collection device or a reliably connected device relaying the data to the cloud.

3.2.2 semantic gateway: A software system that is used for conversion between various IoT protocols, applications and services and is included in heterogeneous gateway systems.

3.2.3 IoT/mobile device: A data-producing device that lacks reliable communication links (IoT, mobile, disaster management, etc.).

4 Abbreviations and acronyms

This Recommendation uses the following abbreviations and acronyms:

AMQP	Advanced Message Queuing Protocol
CBOR	Concise Binary Object Representation
CoAP	Constrained Application Protocol
CRDT	Conflict-free Replicated Data Type
GI	Gateway Infrastructure
HTTP	Hypertext Transfer Protocol
ICF	Intermediary Conversion Format
IoT	Internet of Things
JSON	JavaScript Object Notation
MQTT	Message Queuing Telemetry Transport
RAM	Random Access Memory
RON	Replicated Object Notation
RTPS-DDS	Real-Time Publish Subscribe- Data Distribution Service
SHA	Secure Hash Algorithm
SIoTG	Semantic Internet of Things Gateway
UML	Unified Modelling Language
UUID	Universally Unique Identifier
XMPP	Extensible Messaging and Presence Protocol

5 Conventions

None.

6 Network architecture for IoT heterogeneous gateway

Figures 6-1 and 6-2 show the general network architecture of the hardware and software system for heterogeneous IoT gateways.

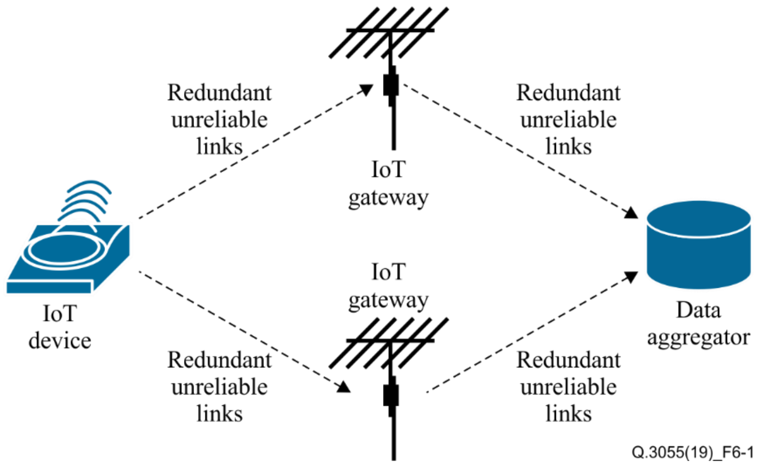

Redundant unreliable links
Redundant unreliable links
IoT gateway
IoT device
IoT gateway
Data aggregator
Redundant unreliable links
Redundant unreliable links
Q.3055(19)_F6-1

Figure 6-1 – General network architecture using heterogeneous IoT gateways

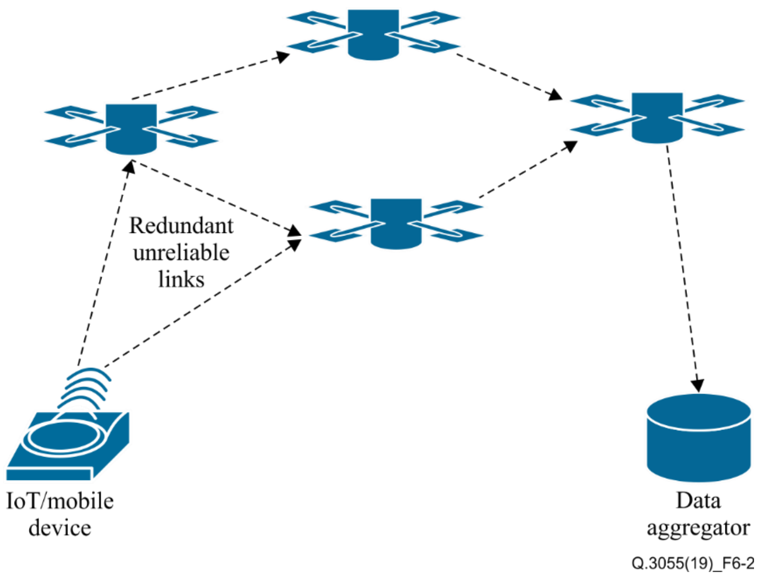


Figure 6-2 – Typical structure of interaction of heterogeneous IoT gateways elements

The system comprises:

– IoT/mobile device: a data-producing device lacking reliable communication links (IoT, mobile, disaster management, etc.).

– Redundant unreliable links: a wireless network with unreliable/fluid topology.

– IoT/mobile gateway: data relay devices, either mobile or unreliably connected:

  • data aggregator: either end-point data collection device or a reliably connected device relaying the data to the cloud.

This network architecture facilitates the implementation of possible interaction scenarios for the heterogeneous IoT gateway and devices connected to it.

7 Semantic gateway architecture

Figure 6-3 shows the UML-model for semantic IoT gateway software.

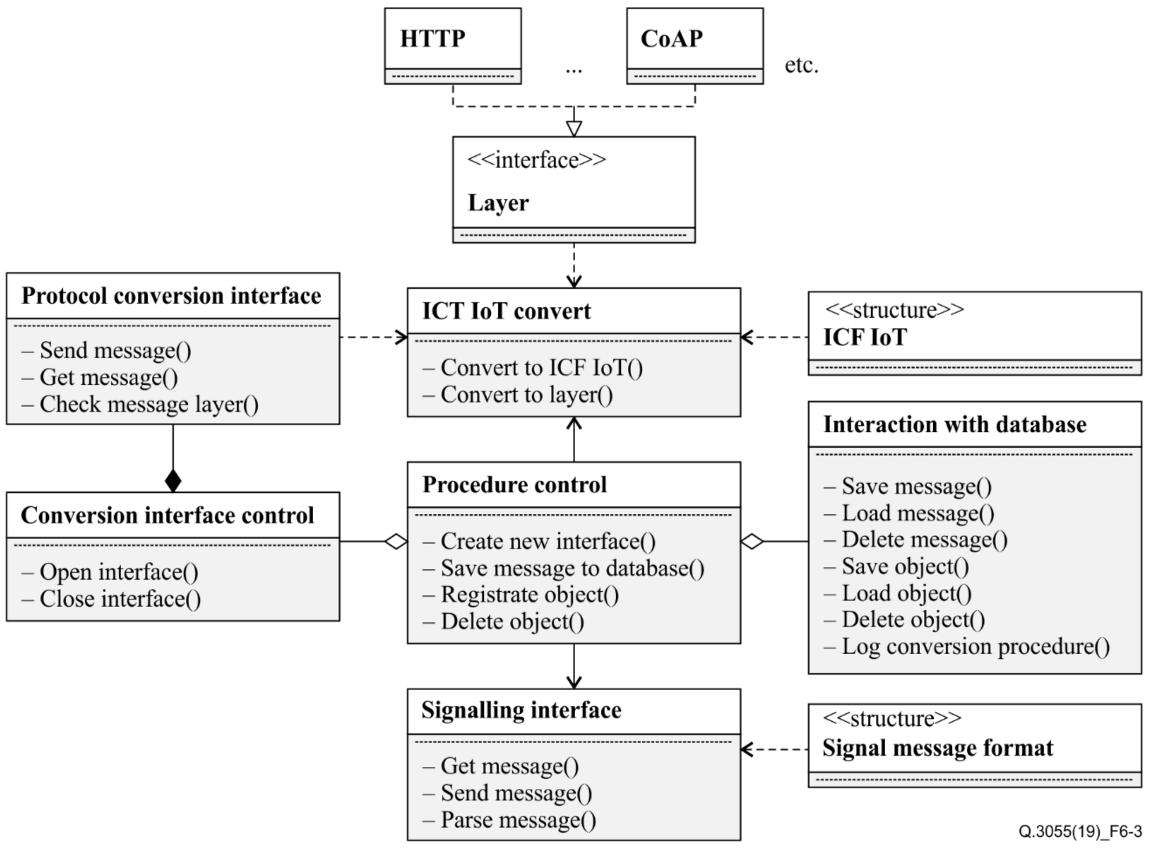


Figure 6-3 – UML-model for semantic IoT gateway software

This model comprises:

Procedure control – this class is responsible for monitoring the implementation of conversion procedures, according to incoming signalling messages.

Signalling interface – this class is responsible for managing the virtual network interface (socket), which receives and processes signalling messages.

Conversion interface control – this class is responsible for managing the virtual network interfaces for IoT protocol conversion.

Protocol conversion interface – this class is responsible for receiving, sending and verifying incoming messages for the IoT protocol conversion interface.

Intermediary conversion format (ICF) IoT convert – this class is responsible for converting messages from/to specific IoT protocol formats to/from intermediary conversion format for IoT.

Interaction with database – this class is responsible for interaction with the local or remote database containing messages for conversion, descriptions of IoT objects (protocols, applications, services) connected to the heterogeneous gateway, and notes on procedure implementation for the semantic gateway (e.g., protocol conversion).

Layer – the interface serving as the basis for the implementation of specific protocols supported by the semantic IoT gateway (e.g., HTTP, CoAP, MQTT, AMQP, RTPS-DDS, XMPP, etc.).

Intermediary conversion format for Internet of things (ICF IoT) – the structure reflecting the intermediary conversion format used for storing information on messages in the database and memory during IoT protocol conversion procedures.

Signal message format – the structure reflecting the format of signalling messages used for storing information on signalling messages during IoT protocol conversion procedures.

8 Database interaction and data collection: Replicated object notation

Replicated object notation (RON) is a data-centric protocol for data delivery/dissemination in an unreliable distributed environment. RON puts an emphasis on mitigating inconsistencies introduced by unreliable heterogeneous environments, such as arbitrary delivery paths, data loss, duplication and reordering, arbitrary delays, scattered and/or mobile sources and/or receivers among others.

RON is data-centric event-sourced format with a simplified tabular structure. RON relies on universally unique identifiers (UUIDs) to resolve inconsistencies and assemble the full state of separately arriving pieces of partial information. The internal structure of the protocol is simple and uniform to ease implementation and use.

Any RON construct is composed of four types of "atoms": strings/buffers, integers, floats and UUIDs. A basic unit of exchange is an immutable "op" corresponding to an atomic event of data change. An op has four metadata UUIDs (data type, object id, event id and a "reference") and an arbitrary number of payload atoms.

A transactional unit of transmission is a frame, which is a sequence of ops.

A library of data recovery/reconciliation algorithms relies on the op metadata to resolve ambiguities and reassemble the data at a collection point.

RON data is effectively independent in the context of any particular transmission or storage. Hence, RON effectively blurs the border between storage/caching and delivery/transmission. Only the eventual propagation of data matters. All means of storage are equally valid as means of transmission. Consequently, RON is especially suitable for networks with unstable topology, redundant but unreliable delivery paths and/or unpredictable or extreme delays.

Appendix I

Replicated object notation

(This appendix does not form an integral part of this Recommendation.)

Replicated Object Notation

Replicated Object Notation (RON) is a format for distributed live data. RON's primary mission is continuous data synchronization. A RON object may naturally have any number of replicas, which may synchronize in real-time or intermittently.

JSON, protobuf, and many other formats implicitly assume serialization of separate state snapshots. RON handles state and updates all the same: state is change and change is state. RON includes metadata, versioning and addressing, all built in. Every object, every change, every version has a UUID. Pieces of data reference each other by UUIDs. Every RON data type is a conflict-free replicated data type (CRDT). With RON metadata, state and updates could always be pieced together. It always merges, it always converges.

From another perspective, RON is like a metric system for data. The imperial system employed various usage-based units: foots, lines, furlongs, links, cables, etc. The metric system defines one unit (a meter), then derives other units from that. Similarly, data might be packed into usage-based units: snapshots, logs, chunks, batches, patches. RON defines an immutable *op*, then derives other units from that, in the form of data structures (arrays, maps, sets, etc.) or storage/transmission units (snapshots, batches/patches, logs, etc.).

The following is a simple object serialized in RON:

```
  @1fLDV+biQFvtGV :lww ,
      'id'        '20MF000CUS',
      'type'      'laptop',
      'cpu'       'i7-8850H',
      'display'   '15.6" UHD IPS multi-touch, 400nits',
      'RAM'       '16 GB DDR4 2666MHz',
      'storage'   '512 GB SSD, PCIe-NVME M.2',
      'graphics'  'NVIDIA GeForce GTX 1050Ti 4GB',
  @1fLDk4+biQFvtGV
      'wlan'      'Intel 9560 802.11AC vPro',
     'camera'    'IR & 720p HD Camera with microphone',
 @sha3 'SfiKqD1atGU5xxv1NLp8uZbAcHQDcX~a1HVk5rQFy_nq';
```

Key RON principles are:

- **Immutability** – RON sees data as a collection of immutable timestamped ops. The example above depicts an object state consisting of ten ops (object creation op at line #1, the initial changeset #2 to #8, another changeset of two ops #9/10 and #11). An op may be referenced, transmitted, stored, applied or rolled back, garbage collected, etc. Every RON data structure (array, object, map, set, etc.) is a collection of immutable ops. Similarly, every data storage or transmission unit is made of ops (patch, state, chain, chunk, frame, object graph, log, yarn, etc.).
- **Addressability** of everything. Changes, versions, objects and every piece of data is uniquely identified and globally referenceable. Above, the first op has an id 1fLDV+biQFvtGV, the second one is 1fLDV00001+biQFvtGV, the third

is 1fLDV00002+biQFvtGV and so on (the notation skips incremental ids). The last two ops (#9-10 and #11) belong to a later changeset, so their ids are 1fLDk4+biQFvtGV, 1fLDk40001+biQFvtGV.

NOTE – RON has no notational nesting (no brackets). Instead, data pieces reference each other by UUIDs, thus forming arbitrary graphs.

- **Causality**. Each RON operation explicitly *references* what other op it is based on. No matter how and when the data is obtained, the correct order and location of data pieces can always be reconstructed. As described above, ops form an orderly chain, so references are skipped, except for the object creation op at line #1 which references its data type lww.
- **Efficiency**. RON data is optimized to make metadata overhead bearable. An op is a very fine-grained unit of change. Thus, RON has to optimize per-op metadata overhead in numerous ways. Op ids get skipped if they go incrementally. References are skipped if they point to the previous op (an op chain is a convenient default). For example, the op at line #2 mentions neither its own id (the first plus 1) nor its reference (the first op). The binary variant of RON employs more sophisticated metadata compression techniques. With no abbreviations, the object would look like a tabular log of ops, two metadata UUIDs per op:

```
@1fLDV00000+biQFvtGV  :lww !
@1fLDV00001+biQFvtGV  :1fLDV00000+biQFvtGV 'id'        '20MF000CUS',
@1fLDV00002+biQFvtGV  :1fLDV00001+biQFvtGV 'type'      'laptop',
...
```

- **Integrity**, as ops form a Merkle structure. If necessary, the data is integrity-checked to the last bit, like in git, BitTorrent, BitCoin and other such systems. In the example above, ten ops form a Merkle chain, so the hash of the last op (line #12) covers them all.

RON's vision is swarms of mobile devices communicating over unreliable wireless networks in an untrusted environment.

UUIDs

RON relies heavily on UUIDs to globally and unambiguously address everything it operates with: operations, patches, versions, objects, hashes, etc.

RON employs its own UUID flavours and custom efficient serialization. Unlike RFC 4122, RON UUIDs:

- can be sorted lexicographically,
- can be efficiently compressed,
- can function as lamport clocks,
- can represent human-friendly string constants.

RON UUIDs are serialized as a pair of 64-bit integers encoded with custom base64 encoding:

```
A/LED0000000+XU5eRJ0000
aaaavvvv vvvvvvvv vvvvvvvv vvvvvvvv vvvvvvvv vvvvvvvv vvvvvvvv vvvvvvvv
00eeoooo oooooooo oooooooo oooooooo oooooooo oooooooo oooooooo oooooooo
```

The bit layout is backwards-compatible with RFC 4122 (hijacking the 0 variant, NCS backward compatibility). Third and fourth bits of 9th byte are used to encode version (blue), four most significant bits of 1st byte are used to encode variety (orange). The bulk of the bits are taken by value (green) and origin (violet).

Versions

Two version bits are encoded using $, %, + or – as separator:

- $ for 00: human readable names,
- % for 01: numbers and hashes,
- + for 10: events (Lamport timestamp, and origin),
- – for 11: derived events (same as event).

Varieties

Four variety bits are encoded using single hex digit 0..F followed by a slash /.

Variety flavour is defined by version.

Variety of zero 0 can be omitted.

Varieties for version $ (names):

- 0000: transcendental/hardcoded name (lww, rga) or a scoped name (myvar$gritzko),
- 0001: ISBN (1/978$1400075997,
- 0011: EAN-13 bar code (3/4006381$333931,
- 0100: SI units (4/m, 4/kg,
- 0101: zip codes (5/2628CD$NL, 5/620078$RU,
- 1010: IATA airport code (A/LED,
- 1011: ticker name (B/GOOG$NASDAQ,
- 1100: ISO 4217 currency code (C/USD, C/GBP,
- 1101: short DNS name (D/google$com,
- 1110: E.164 intl phone num (E/7999$5631415,
- 1111: ISO 3166 country code (F/RU, F/FRA...).

Varieties for version % (numbers and hashes):

- 0000..0011: Decimal index (up to 9999999999%, also 2D indices 4%5),
- 0100: SHA-2, plain chunk hash, first 120 bits,
- 0101: SHA-3, plain chunk hash,
- 0110: SHA-2 based RFC 7574 Merkle hash,
- 0111: SHA-3 based RFC 7574 Merkle hash,
- 1000..1011: Random number (A/k3R9w_2F8w%Le~6dDScsw),
- 1100..1111: Crypto id, public key fingerprint.

Varieties for versions + and – (events)

Timestamp type:

- 00__: Base64 calendar (MMDHmSsnn),
- 01__: Logical (40000000001),
- 10__: Epoch (RFC 4122 epoch, 100ns since 1582),

bitwise and with replica id assignment rule:

- __00: trie-forked,
- __01: crypto-forked,
- __10: record-forked,

- __11: application-specific.

UUID compression

Most of RON UUIDs can be efficiently compressed by only encoding highest significant bits.

Base 64×64 tailing zeroes can be omitted:

A/LED0000000+0000000000 = A/LED+0

A/LED0000000$123 = A/LED$123

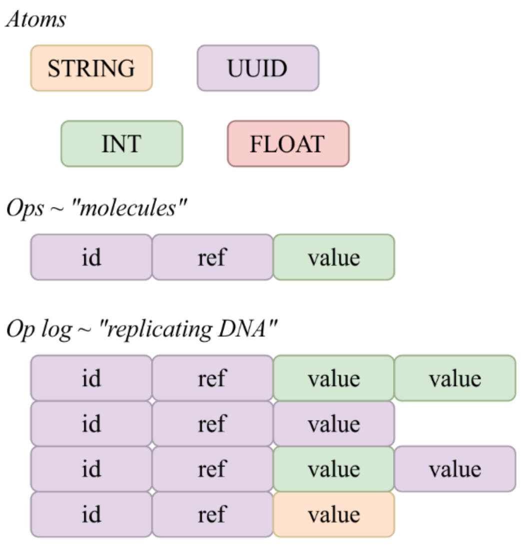
If version is $ and second component is zero, it can be fully omitted:

A/LED0000000$0000000000 = A/LED$0 = A/LED

If variety is 0, it can be omitted as well:

0/lww0000000$0000000000 = lww0000000$0000000000 = lww$0 = lww

RON term glossary



Q.3055(19)_FI.1

- RON UUID – a 128-bit globally unique identifier, one of four *versions*:
  - time-based (a logical/hybrid timestamp, 60 bits of timestamp, 60 bits of event *origin* id)
  - name (a human readable name of some predefined concept, e.g., a data type or an error type)
  - numeric (either an arbitrary number or a hash)
- Atom – an immutable value of one of four types:
  - RON UUID
  - Integer (64-bit signed)
  - String (UTF-8)
  - Floating-point number (IEEE 754-2008, 64 bit)
- Op – an immutable unit of change, consists of:
  - specifier (metadata)
    - own RON UUID (id, identifies the op)
    - reference RON UUID (ref, identifies the op's location in the data graph)

  - value (payload data)
    - any number (zero or more) of value atoms (UUIDs, ints, strings, floats)

If RON atoms are compared to real (chemistry) atoms, then RON ops are "molecules", composed of "atoms". Then, an op log is our "DNA" – it contains all the replicated data. Other higher-value constructs are like polypeptides, proteins and suchlike – they are composed of ops. Fundamentally, they are either subsets or projections of the op log (the "DNA"). A replicated op log is the foundation of all this machinery.

Op-collection construct

- Chain – a sequence of ops from the same origin, where each next op references the previous one.
- Span – (chain span) – a chain where each op's id is exactly an increment of the previous id (1gABC+origin, 1gABC00001+origin, 1gABC00002+origin...).
- Yarn – a linear log of all ops from the same origin (corresponds to a Lamport process).
- Tree – a causally ordered group of ops forming a tree (each next op references some preceding op from the tree, except for the root op).
- Object – largely synonymous to a tree, although op ordering depends on the data type (RDT).
- Header – the object creation op (its id becomes the object's id; the header op is the root of the object's op tree).
- Patch – a group of ops modifying the same tree/object (causally consistent, i.e., referencing the existing ops of the tree or previous ops of the patch).
- Frame – largely synonymous to a "write batch"; a group of ops to be applied atomically, in a single transaction.
- Chunk – a group of object's ops, preceded by the object's header, e.g., object state or a patch.
- Log – a causally ordered sequence of ops, like a database op log. While yarns are linearly ordered, a log only has partial (causal) order; different replicas of a log may go in slightly different orders.
- Graph – a group of objects referencing each other.
- Graph patch – a group of object patches and full object states, a causally consistent change of an object graph.
- Segments – a log segment, a yarn segment, a chain segment, etc. – a subset of the construct retaining its key features.

Other terms

- Annotation – a pseudo-op that is not itself a data change, but some derived/secondary information, related to some op (e.g., its hash or other metadata).
- Vector timestamp – an array of time-based UUIDs, one per origin; a timestamp produced by vector clocks.

Nominal format

Nominal RON is a simple memory layout that stores uncompressed RON ops. It serves three purposes:

- exchange RON ops within the same address space,
- define the canonical representation, e.g., content hashing,
- serve as an intermediary format in any conversions, mappings, filters and transformations.

Normally, a serialized RON frame is read by an iterator/parser/cursor which creates a nominal-RON representation of each next op. A builder/serializer/writer converts a nominal-RON op into the resulting format.

The nominal format per se is not intended as a frame serialization format, as it would be too inefficient.

The nominal RON may be highly beneficial in mixed environments, e.g., when a C++ implementation is used from a node.js program through the bindings. If the js part may consume/produce nominal-RON data, it needs no own parsers and no own builders. Instead, it may consume parsed data from a scratchpad buffer. The same applies to bizzare situations, e.g., C++ engine/parsers/builders running in a browser under wasm, used by a Java program running under GWT.

Overall layout

The nominal format is made of 128-bit atoms, consisting of two 64-bit words each. Words use the 4+60 bit layout. Nominally, the byte layout is big-endian (e.g., a hash function expects big-endian words). Although, an implementation may use little-endian words if requested. All pictures assume big-endian words.

An op is a concatenation of its atoms, starting with its id and ref UUIDs.

In all atoms except UUIDs, the origin word specifies a byte range in the original RON buffer (text RON, binary, JSON, CBOR). The maximum allowed frame size is 1 GB, so the byte range is given as two 30-bit unsigned ints for the offset and the length. The raw buffer is supposed to be available.

Caveats:

- String values only reside in the raw buffer.
- UUID atoms are fully parsed, so no range provided.

Frame header

(rarely used)

Bytes 0-3 (mmmm) contain magic bytes, RON2 for closed RON, ROP2 for open.

Bytes 4-7 (cccc) is the number of ops in the frame, as a big-endian integer.

Bits 4-7 in byte 8 are set to 1100.

Bytes 8-15 contain the byte range within the raw unparsed buffer.

```
mmmmmmmm mmmmmmmm mmmmmmmm mmmmmmmm cccccccc cccccccc cccccccc cccccccc
1100oooo oooooooo oooooooo oooooooo oollllll llllllll llllllll llllllll
```

Again, frame header is supposed to be rarely used, as entire frames are rarely uncompressed at once.

Op header

(less rarely used)

Bytes 0-3 (mmmm) contain magic bytes, ron2 for closed RON, rop2 for open.

Bytes 4-7 (cccc) contain the number of atoms in the op, as a big-endian integer.

Bits 4-7 in byte 8 (11__) indicate the op type:

- 1100 raw,
- 1101 reduced,
- 1110 header,
- 1111 query.

Bytes 8-15 contain location of *raw buffer* location with unparsed op content.

```
mmmmmmmm mmmmmmmm mmmmmmmm mmmmmmmm cccccccc cccccccc cccccccc cccccccc
11__oooo oooooooo oooooooo oooooooo oollllll llllllll llllllll llllllll
```

UUID atom

A 128-bit RON UUID:

```
vvvv···· ········ ········ ········ ········ ········ ········ ·······
00ss···· ········ ········ ········ ········ ········ ········ ·······
```

Bits 4-7 in byte 0 (vvvv) represent UUID *variety*.

Bits 6-7 in byte 8 are set to 00, indicating that this atom is a UUID.

Bits 4-5 in byte 8 (ss) represent UUID *version*.

(That is the usual RON UUID in-memory layout, except implementations that are likely to use platform endiannes internally.)

Integer atom

Bytes 0-7 (iiii) a 4-bit signed integer.

Bytes 8-15 contain location of *raw buffer* location with unparsed integer representation.

Bits 4-7 in byte 8 are set to 0101, indicating that this atom is integer atom.

60 value bits of the origin word (ooo, lll) contain the raw buffer range for the original unparsed value. ooo is the offset, lll is the length, both are 30-bit unsigned integers.

```
iiiiiiii iiiiiiii iiiiiiii iiiiiiii iiiiiiii iiiiiiii iiiiiiii iiiiiiii
0101oooo oooooooo oooooooo oooooooo oollllll llllllll llllllll llllllll
```

Float atom

Bytes 0-7 (ffff) contain a 64-bit float (should be IEEE 754).

Bits 4-7 in byte 8 are set to 0111, indicating that this atom is a float.

Bytes 8-15 contain a *raw buffer* range for the unparsed float representation, same as integer.

```
ffffffff ffffffff ffffffff ffffffff ffffffff ffffffff ffffffff ffffffff
0111oooo oooooooo oooooooo oooooooo oollllll llllllll llllllll llllllll
```

String atom

The value word contains string metrics: byte length bbb and codepoint length ccc. These metrics are relative to a pure UTF-8 string, all escapes etc., replaced.

Bits 4-7 in byte 8 are set to 0110, indicating that this atom is a string.

Bytes 8-15 contain a *raw buffer* range (note the raw byte length may not match the UTF-8 byte length).

```
0000bbbb bbbbbbbb bbbbbbbb bbbbbbbb bbcccccc cccccccc cccccccc cccccccc
0110oooo oooooooo oooooooo oooooooo oollllll llllllll llllllll llllllll
```

Text format (Grammar)

Text-based RON is a regular language, mostly used for its human readability.

The Ragel grammar for RON UUID is:

```
# digits (base64, hex)
 DGT = [0-9a-zA-Z~_];
 HEX = [0-9A-F];

 # RON UUID variety (1st word flag bits)
 VARIETY = HEX "/" @variety;

 # RON UUID version char (2nd word flag bits)
 VERSION = [\$\%\+\-] @version;

 # 60+4 bit word (the most significant 4 bits are flags, the rest is payload)
 WORD = DGT+;

 # UUID value (the 1st word)
 VALUE = WORD >begin_value %end_value;

 # UUID origin (the 2nd word)
 ORIGIN = WORD >begin_origin %end_origin;

 # RON UUID (128 bits, two 60+4 bit words)
 UUID = ( VARIETY? VALUE ( VERSION ORIGIN )? ) >begin_uuid %end_uuid;
 # examples: lww, A/LED, 12345+origin, 1/0000000001+origin, some_hash%12345375868
```

The Ragel grammar for the text-based RON is:

```
# int64_t
 SGN = [\-+];
 DIGITS = digit+;
 INT = (SGN? DIGITS ) >begin_int %end_int;

 # 64-bit (double) float TODO ISO syntax
 FRAC = "." DIGITS;
 EXP = [eE] SGN? DIGITS;
FLOAT = ( SGN? DIGITS ( FRAC EXP? | EXP ) ) >begin_float %end_float;

 # a char TODO UTF8, escapes, \u escapes
 # UTF8 = TODO;

 # JSON-ey string
 UNIESC = "\\u" [0-9a-fA-F]{4};
 ESC = "\\" [nrt\\b'/"];
 CHAR = CODEPOINT - ['\n\r\\];
```

```
    STRING = ( (UNIESC|ESC|CHAR)* ) >begin_string %end_string;

    # op term (header op, raw/reduced op, query op)
    OPTERM = [,;!?] @op_term;

    # value atom (payload) – int, float, string, UUID
    BARE_ATOM = INT | FLOAT | UUID %end_bare_uuid;
    QUOTED_ATOM =
            "=" space* INT  |
            "^" space* FLOAT |
            ['] STRING ['] |
            ">" space* UUID %end_quoted_uuid ;
    ATOM = QUOTED_ATOM | space BARE_ATOM ;

    # op's specifier, @id :ref
    SPEC = '@' UUID %end_id space* ( ':' UUID %end_ref )? ;
    ATOMS = ATOM (space* ATOM)* ;

    # RON op: an immutable unit of change
    OP = (SPEC|BARE_ATOM)? space* ATOMS? space* OPTERM ;

    # a frame terminator (mandatory in the streaming mode)
    DOT = ".\n" ;

    # RON frame (open text coding)
    TEXT_FRAME = (space* OP)* space* DOT? ;
```

Binary format

The binary RON format is more efficient because of higher bit density. It is also simpler and safer to parse because of explicit field lengths. Obviously, it is not human-readable.

Like the text format, the binary one is only optimized for iteration. As a result of compression, records are inevitably of variable length, so random access is not possible. Also, compression depends on iteration, as UUIDs get abbreviated relative to similar preceding UUIDs.

A binary RON frame starts with magic bytes RON2 and frame length. The rest of the frame is a sequence of *fields*. Each field starts with a *descriptor* specifying the type of the field and its length.

Frame length is serialized as a 32-bit big-endian integer. The maximum length of a frame is 2^{30} bytes (a gibibyte). If the length value has its most significant bit set to 1, then the frame is *chunked*. A chunked frame is followed by a continuation frame. A continuation frame has no magic bytes, just a 4-byte length field. The last continuation frame must have the m.s.b. of its length set to 0.

A descriptor's first byte spends four most significant (m.s.) bits to describe the type of the field, other four bits describe its length.

128	64	32	16	8	4	2	1
Major type		Minor type		Field length			

Field descriptor major/minor type bits are set as follows:

1) 00 RON op descriptor,
   - 0000 raw op,
   - 0001 reduced op,
   - 0010 header op,
   - 0011 query header op.
2) 01 Reserved (for binary data)
   - 0100 type (reducer) id,
   - 0101 object id,
   - 0110 event id,
   - 0111 ref/location id
3) 10 Atoms, compressed (zipped chains)
   - 1000 UUID, backreference
   - 1001 integer list
   - 1010 char list
   - 1011
4) 11 Atom
   - 1100 UUID, uncompressed (lengths 1..16)
   - 1101 integer (big-endian, zigzag-coded, lengths 1, 2, 4, 8)
   - 1110 string (UTF-8, length 0..$2^{31}-1$)
   - 1111 float (IEEE 754-2008, binary 16, 32 or 64, lengths 2, 4, 8 resp)

A descriptor's four least significant bits encode the length of the field in question. The length value given by a descriptor does not include the length of the descriptor itself.

If a field or a frame is 1 to 16 bytes long, then it has its length coded directly in the four l.s. bits of the descriptor. Zero stands for the length of 16 because most field types are limited to that length. Op terms specify no length. With string atoms, zero denotes the presence of an extended length field which is either 1 or 4 bytes long. The maximum allowed string length is 1Gb (30 bits). In case the descriptor byte is exactly 1110 0000, the m.s. bit of the next byte denotes the length of the extended length field (0 for one, 1 for four bytes). The rest of the next byte (and possibly other three) is a big-endian integer denoting the byte length of the string.

Consider a time value query frame: *now?.

- 4 bytes are magic bytes (RON, 0101 0010 0100 1111 0100 1110 0011 0010)
- frame length: 4 bytes (length 5, 0000 0000 0000 0000 0000 0000 0000 0101)
- op term descriptor: 1 byte (0011 0000)
- uncompressed UUID descriptor: 1 byte (cited length 3, 0100 0011)
- now RON UUID: 3 bytes (0000 1100 1011 0011 1110 1100, the "uncompressed" coding still trims a lot of zeroes, see below).

As UUID length is up to 16 bytes, UUID fields never use a separate length number. UUID descriptors are always 1 byte long. The length of 0 stands for 16.

Length bits 0000 stand for:

- zero length for op terms,
- 16 for integer/float atoms, zipped/unzipped UUIDs,
- for strings, that signals an extended length record (1 or 4 bytes).

An extended length record is used for strings, as those can be up to 2 GB long. An extended length record is either 1 or four bytes. Four-byte record is a big-endian 32-bit int having its m.s. bit set to 1. Thus, strings of 127 bytes and shorter may use 1 byte long length record.

Ops

Op term fields may have cited length of 0000 or be skipped if they match the previous op's term. Nevertheless, sometimes it is necessary to introduce redundancy, CRC/checksumming, hashing, etc. Exactly for this purpose non-empty terms may be used. The checksumming method is specified by the field length (TODO).

Atoms

Strings are serialized as UTF-8.

Integers are serialized using the zig-zag coding (the l.s. bit conveys the sign).

Floats are serialized as IEEE 754 floats (4-byte and 8-byte support is required, other lengths are optional).

Appendix II

RON use case

(This appendix does not form an integral part of this Recommendation.)

II.1 Data bees

The use case envisions drones collect data from geographically distributed sensors, e.g., in agriculture. Solar-powered sensors have limited power capacity, hence only able to carry out short-distance radio communication. There is no requirement of real-time data delivery. Still, the complete time series data has to be collected with a reasonable lag. No wired infrastructure is feasible in this case. Radio-relay networks are also difficult to maintain in the given circumstances.

The solution is to use "data bees" – drones flying over the area and collecting the data using high-bandwidth short-range communication. The sensor data is accumulated in the drone's memory. Once the drone is back from the fields, the data is uploaded to the database. Sensors may not be aware of their coordinates or even current date/time. Such an information may be added by the drone. Drones launched from a mobile platform (e.g., a car) introduce an additional leg of data transfer.

As all the devices and all RON events have globally unique identifiers, the data is independent of its collection procedure or place of storage, can be freely annotated and cross-referenced, matched with data from other sources. The event-based nature of RON allows for incremental synchronization.

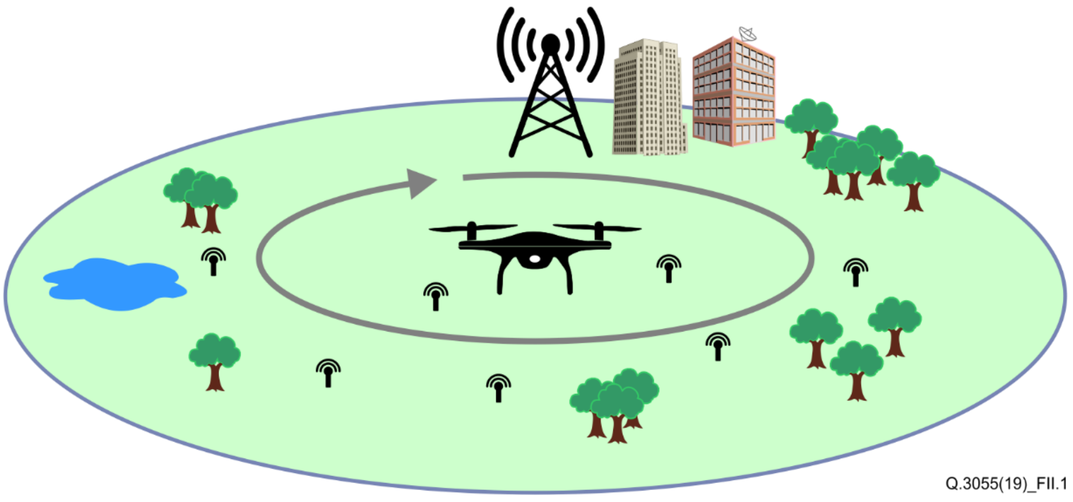


Figure II.1 – Data bee

II.2 RON use case: vehicle fleet telemetry

The use case envisions a fleet of vehicles producing non-trivial amounts of telemetry data. The data cannot be transferred by air due to lack of coverage, lack of bandwidth and/or lack of urgency. Nevertheless, the data can be recorded locally, as a vehicle's data storage capacity is practically unlimited. Some parts of the data may be uploaded by the wireless connection in real-time, some by a wired connection during maintenance, some only inspected locally, or never inspected.

RON relies on globally unique part identifiers (64 bit) used to produce globally unique event identifiers (128 bit). The latter allow to address/reference any telemetry event, globally. Independent of the particular route the data had to take, there is no confusion possible. Telemetry events may reference each other, e.g., a high-level aggregate referencing sensor data of its parts or a vehicle referencing the data from other vehicles.

Uploading complete datasets might turn unfeasible. The data might as well be processed locally, with derived data referencing the original data. Another option for telemetry upload is simply sending the storage device by regular mail. Despite the intermittent connectivity of vehicular networks and all the various paths the data might take, RON data forms a single network, a "Web of machines".

SERIES OF ITU-T RECOMMENDATIONS

Series A	Organization of the work of ITU-T
Series D	Tariff and accounting principles and international telecommunication/ICT economic and policy issues
Series E	Overall network operation, telephone service, service operation and human factors
Series F	Non-telephone telecommunication services
Series G	Transmission systems and media, digital systems and networks
Series H	Audiovisual and multimedia systems
Series I	Integrated services digital network
Series J	Cable networks and transmission of television, sound programme and other multimedia signals
Series K	Protection against interference
Series L	Environment and ICTs, climate change, e-waste, energy efficiency; construction, installation and protection of cables and other elements of outside plant
Series M	Telecommunication management, including TMN and network maintenance
Series N	Maintenance: international sound programme and television transmission circuits
Series O	Specifications of measuring equipment
Series P	Telephone transmission quality, telephone installations, local line networks
Series Q	**Switching and signalling, and associated measurements and tests**
Series R	Telegraph transmission
Series S	Telegraph services terminal equipment
Series T	Terminals for telematic services
Series U	Telegraph switching
Series V	Data communication over the telephone network
Series X	Data networks, open system communications and security
Series Y	Global information infrastructure, Internet protocol aspects, next-generation networks, Internet of Things and smart cities
Series Z	Languages and general software aspects for telecommunication systems

Printed in Switzerland
Geneva, 2020