XEP-0323: Internet of Things - Sensor Data

Peter Waher
mailto:peterwaher@hotmail.com
xmpp:peter.waher@jabber.org
http://www.linkedin.com/in/peterwaher

2017-05-20
Version 0.6

Status	Type	Short Name
Retracted	Standards Track	sensor-data

Note: This specification has been retracted by the author; new implementations are not recommended.

This specification provides the common framework for sensor data interchange over XMPP networks.

Legal

Copyright

This XMPP Extension Protocol is copyright © 1999 – 2024 by the XMPP Standards Foundation (XSF).

Permissions

Permission is hereby granted, free of charge, to any person obtaining a copy of this specification (the ”Specification”), to make use of the Specification without restriction, including without limitation the rights to implement the Specification in a software program, deploy the Specification in a network service, and copy, modify, merge, publish, translate, distribute, sublicense, or sell copies of the Specification, and to permit persons to whom the Specification is furnished to do so, subject to the condition that the foregoing copyright notice and this permission notice shall be included in all copies or substantial portions of the Specification. Unless separate permission is granted, modified works that are redistributed shall not contain misleading information regarding the authors, title, number, or publisher of the Specification, and shall not claim endorsement of the modified works by the authors, any organization or project to which the authors belong, or the XMPP Standards Foundation.

Warranty

NOTE WELL: This Specification is provided on an ”AS IS” BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, express or implied, including, without limitation, any warranties or conditions of TITLE, NON-INFRINGEMENT, MERCHANTABILITY, or FITNESS FOR A PARTICULAR PURPOSE.

Liability

In no event and under no legal theory, whether in tort (including negligence), contract, or otherwise, unless required by applicable law (such as deliberate and grossly negligent acts) or agreed to in writing, shall the XMPP Standards Foundation or any author of this Specification be liable for damages, including any direct, indirect, special, incidental, or consequential damages of any character arising from, out of, or in connection with the Specification or the implementation, deployment, or other use of the Specification (including but not limited to damages for loss of goodwill, work stoppage, computer failure or malfunction, or any and all other commercial damages or losses), even if the XMPP Standards Foundation or such author has been advised of the possibility of such damages.

Conformance

This XMPP Extension Protocol has been contributed in full conformance with the XSF’s Intellectual Property Rights Policy (a copy of which can be found at <https://xmpp.org/about/xsf/ipr-policy> or obtained by writing to XMPP Standards Foundation, P.O. Box 787, Parker, CO 80134 USA).

Contents

1 Introduction 1

2 Glossary 2

3 Use Cases 4
- 3.1 Request Read-out of momentary values . . . 6
- 3.2 Read-out failure . . . 7
- 3.3 Read-out rejected . . . 8
- 3.4 Read-out all . . . 9
- 3.5 Read-out of multiple devices . . . 10
- 3.6 Read-out of specific fields . . . 11
- 3.7 Cancelling a scheduled read-out request . . . 12

4 Determining Support 13

5 Implementation Notes 14
- 5.1 String lengths . . . 14
- 5.2 Enumerations vs. Strings . . . 14
- 5.3 Asynchronous feedback . . . 15
- 5.4 Field Value Data Types . . . 15
- 5.5 Harmonization with XEP-0325 (Control) . . . 16
- 5.6 Field Types . . . 17
- 5.7 Field Quality of Service Values . . . 18
  - 5.7.1 Estimates vs. Readouts . . . 19
- 5.8 Subnodes and supernodes . . . 20
- 5.9 Reading devices from large subsystems . . . 20
- 5.10 Reading controllable parameter values . . . 21

6 Internationalization Considerations 21
- 6.1 Time Zones . . . 21
- 6.2 Localized strings . . . 21

7 Security Considerations 24

8 IANA Considerations 24

9 XMPP Registrar Considerations 24

10 XML Schema 24

11 For more information 31

12 Acknowledgements 31

1 Introduction

This XEP provides the underlying architecture, basic operations and data structures for sensor data communication over XMPP networks. It includes a hardware abstraction model, removing any technical detail implemented in underlying technologies.
Note has to be taken, that these XEP's are designed for implementation in sensors, many of which have very limited amount of memory (both RAM and ROM) or resources (processing power). Therefore, simplicity is of utmost importance. Furthermore, sensor networks can become huge, easily with millions of devices in peer-to-peer networks.
Sensor networks contains many different architectures and use cases. For this reason, the sensor network standards have been divided into multiple XEPs according to the following table:

XEP	Description
xep-0000-IoT-BatteryPoweredSensors	Defines how to handle the peculiars related to battery powered devices, and other devices intermittently available on the network.
xep-0000-IoT-Events	Defines how sensors send events, how event subscription, hysteresis levels, etc., are configured.
xep-0000-IoT-Interoperability	Defines guidelines for how to achieve interoperability in sensor networks, publishing interoperability interfaces for different types of devices.
xep-0000-IoT-Multicast	Defines how sensor data can be multicast in efficient ways.
xep-0000-IoT-PubSub	Defines how efficient publication of sensor data can be made in sensor networks.
xep-0000-IoT-Chat	Defines how human-to-machine interfaces should be constructed using chat messages to be user friendly, automatable and consistent with other IoT extensions and possible underlying architecture.
XEP-0322	Defines how to EXI can be used in XMPP to achieve efficient compression of data. Albeit not a sensor network specific XEP, this XEP should be considered in all sensor network implementations where memory and packet size is an issue.
XEP-0323	This specification. Provides the underlying architecture, basic operations and data structures for sensor data communication over XMPP networks. It includes a hardware abstraction model, removing any technical detail implemented in underlying technologies. This XEP is used by all other sensor network XEPs.

XEP	Description
XEP-0324	Defines how provisioning, the management of access privileges, etc., can be efficiently and easily implemented.
XEP-0325	Defines how to control actuators and other devices in Internet of Things.
XEP-0326	Defines how to handle architectures containing concentrators or servers handling multiple sensors.
XEP-0331	Defines extensions for how color parameters can be handled, based on Data Forms (XEP-0004) XEP-0004: Data Forms <https://xmpp.org/extensions/xep-0004.html>.
XEP-0336	Defines extensions for how dynamic forms can be created, based on Data Forms (XEP-0004) XEP-0004: Data Forms <https://xmpp.org/extensions/xep-0004.html>., Data Forms Validation (XEP-0122) XEP-0122: Data Forms Validation <https://xmpp.org/extensions/xep-0122.html>., Publishing Stream Initiation Requests (XEP-0137) XEP-0137: Publishing Stream Initiation Requests <https://xmpp.org/extensions/xep-0137.html>. and Data Forms Layout (XEP-0141) XEP-0141: Data Forms Layout <https://xmpp.org/extensions/xep-0141.html>..
XEP-0347	Defines the peculiars of sensor discovery in sensor networks. Apart from discovering sensors by JID, it also defines how to discover sensors based on location, etc.

2 Glossary

The following table lists common terms and corresponding descriptions.

Actuator Device containing at least one configurable property or output that can and should be controlled by some other entity or device.

Computed Value A value that is computed instead of measured.

Concentrator Device managing a set of devices which it publishes on the XMPP network.

Field One item of sensor data. Contains information about: Node, Field Name, Value, Precision, Unit, Value Type, Status, Timestamp, Localization information, etc. Fields should be unique within the triple (Node ID, Field Name, Timestamp).

Field Name Name of a field of sensor data. Examples: Energy, Volume, Flow, Power, etc.

Field Type What type of value the field represents. Examples: Momentary Value, Status Value, Identification Value, Calculated Value, Peak Value, Historical Value, etc.

Historical Value A value stored in memory from a previous timestamp.

Identification Value A value that can be used for identification. (Serial numbers, meter IDs, locations, names, etc.)

Localization information Optional information for a field, allowing the sensor to control how the information should be presented to human viewers.

Meter A device possible containing multiple sensors, used in metering applications. Examples: Electricity meter, Water Meter, Heat Meter, Cooling Meter, etc.

Momentary Value A momentary value represents a value measured at the time of the read-out.

Node Graphs contain nodes and edges between nodes. In Internet of Things, sensors, actuators, meters, devices, gateways, etc., are often depicted as nodes whereas links between sensors (friendships) are depicted as edges. In abstract terms, it's easier to talk about a Node, rather than list different possible node types (sensors, actuators, meters, devices, gateways, etc.). Each Node has a Node ID.

Node ID An ID uniquely identifying a node within its corresponding context. If a globally unique ID is desired, an architecture should be used using a universally accepted ID scheme.

Parameter Readable and/or writable property on a node/device. The XEP-0326 Internet of Things - Concentrators (XEP-0326) XEP-0326: Internet of Things - Concentrators <https://xmpp.org/extensions/xep-0326.html>. deals with reading and writing parameters on nodes/devices. Fields are not parameters, and parameters are not fields.

Peak Value A maximum or minimum value during a given period.

Precision In physics, precision determines the number of digits of precision. In sensor networks however, this definition is not easily applicable. Instead, precision determines, for example, the number of decimals of precision, or power of precision. Example: 123.200 MWh contains 3 decimals of precision. All entities parsing and delivering field information in sensor networks should always retain the number of decimals in a message.

Sensor Device measuring at least one digital value (0 or 1) or analog value (value with precision and physical unit). Examples: Temperature sensor, pressure sensor, etc. Sensor values are reported as fields during read-out. Each sensor has a unique Node ID.

SN Sensor Network. A network consisting, but not limited to sensors, where transport and use of sensor data is of primary concern. A sensor network may contain actuators, network applications, monitors, services, etc.

Status Value A value displaying status information about something.

Timestamp Timestamp of value, when the value was sampled or recorded.

Token A client, device or user can get a token from a provisioning server. These tokens can be included in requests to other entities in the network, so these entities can validate access rights with the provisioning server.

Unit Physical unit of value. Example: MWh, l/s, etc.

Value A field value.

Value Status Status of field value. Contains important status information for Quality of Service purposes. Examples: Ok, Error, Warning, Time Shifted, Missing, Signed, etc.

Value Type Can be numeric, string, boolean, Date & Time, Time Span or Enumeration.

WSN Wireless Sensor Network, a sensor network including wireless devices.

XMPP Client Application connected to an XMPP network, having a JID. Note that sensors, as well as applications requesting sensor data can be XMPP clients.

3 Use Cases

The most common use case for a sensor network application is meter read-out. It's performed using a request and response mechanism, as is shown in the following diagram.

The read-out request is started by the client sending a **req** request to the device. Here, the client selects a sequence number **seqnr**. It should be unique among requests made by the client. The device will use this sequence numbers in all messages sent back to the client.
The request also contains a set of **field types** that very roughly determine what the client wants to read. What the client actually will return will be determined by a lot of other factors, such as make and model of device, any provisioning rules provided, etc. This parameter just gives a hint on what kind of data is desired. It is implicit in the request by the context what kind of data is requested. Examples of field types are: Momentary values, peak values, historical values, computed values, status values, identification values, etc.
If reading historical values, the client can also specify an optional time range using the **from** and **to** parameter values, giving the device a hint on how much data to return.
If the client wants the read-out to be performed at a given point in time, the client can define this using the optional parameter **when**.

There's an optional parameter **ids** that the client can provide, listing a set of **Node IDs**. If omitted, the request includes all sensors or devices managed by the current JID. But, if the JID is controlled by a system, device or concentrator managing various devices, the **ids** parameter restricts the read-out to specific individuals.
Note: The device is not required to follow the hints given by the client. These are suggestions the client can use to minimize its effort to perform the read-out. The client MUST make sure the response is filtered according to original requirements by the client after the read-out response has been received.
If the device accepts the client request, it sends an **accepted** response back to the client. The device also has to determine if the read-out is commenced directly, or if it is to be queued for later processing. Note that the request can be queued for several reasons. The device can be busy, and queues it until it is ready to process the request. It can also queue the request if the client has requested it to be executed at a given time. If the request is queued, the device informs the client of this using the **queued** attribute. Note however, that the device will process the request when it can. There's no guarantee that the device will be able to process the request exactly when the client requests it.
Note: The **accepted** message can be omitted if the device already has the response and is ready to send it. If the client receives field data or a **done** message before receiving an **accepted** message, the client can assume the device accepted the request and omitted sending an **accepted** element.
If the request was queued, the device will send a message informing the client when the read-out is begun. This is done using a **started** message, using the same **seqnr** used in the original request.
Note: Sending a **started** element should be omitted by the device if the request is not queued on the device. If the **queued** attribute is omitted in the response, or has the value **false**, the client must not assume the device will send a **started** element.
During the read-out, the device sends partial results back to the client using the same **seqnr** as used in the request, using a **fields** message. These messages will contain a sequence of fields read out of the device. The client is required to filter this list according to original specifications, as the device is not required to do this filtering for the client.
When read-out is complete, the device will send a **done** message to the client with the same **seqnr** as in the original request. Since the sender of messages in the device at the time of sending might not be aware of if there are more messages to send or not, the device can send this message separately as is shown in the diagram. If the device however, knows the last message containing fields is the last, it can set a **done** attribute in the message, to skip this last message.
Note: There is no guarantee that the device will send a corresponding **started** and **fields** element, even though the request was accepted. The device might lose power during the process and forget the request. The client should always be aware of that devices may not respond in time, and take appropriate action accordingly (for instance, implementing a retry mechanism).
If a failure occurs while performing the read-out, a **failure** message is sent, instead of a corresponding **fields** message, as is shown in the following diagram. Apart from notifying the client that a failure to perform the read-out, or part thereof, has occurred, it also provides a

list of errors that the device encountered while trying. Note that multiple **fields** and **failure** messages can be sent back to the client during the read-out.

The device can also reject a read-out request. Reasons for rejecting a request may be missing privileges defined by provisioning rules, etc. It's not part of this XEP to define such rules. A separate XEP (Internet of Things - Provisioning (XEP-0324) [1]) defines an architecture for how such provisioning can be easily implemented.
A rejection response is shown in the following diagram.

If a read-out has been queued, the client can cancel the queued read-out request sending a **cancel** command to the device. If a reading has begin and the client sends a **cancel** command to the device, the device can choose if the read-out should be cancelled or completed.
Note: Remember that the **seqnr** value used in this command is unique only to the client making the request. The device can receive requests from multiple clients, and must make sure it differs between **seqnr** values from different clients. Different clients are assumed to have different values in the corresponding **from** attributes.

3.1 Request Read-out of momentary values

The client that wishes to receive momentary values from the sensor initiates the request using the **req** element sent to the device.

Listing 1: Read-out request of momentary values from a device

```
<iq type='get'
  from='client@example.org/amr'
  to='device@example.org'
  id='S0001'>
  <req xmlns='urn:xmpp:iot:sensordata' seqnr='1' momentary='true'/>
</iq>
```

When the device has received and accepted the request, it responds as follows:

Listing 2: Read-out request accepted by device

[1] XEP-0324: Internet of Things - Provisioning <https://xmpp.org/extensions/xep-0324.html>.

```
<iq type='result'
  from='device@example.org'
  to='client@example.org/amr'
  id='S0001'>
  <accepted xmlns='urn:xmpp:iot:sensordata' seqnr='1'/>
</iq>
```

When read-out is complete, the response is sent as follows:

Listing 3: Momentary read-out response

```
<message from='device@example.org'
  to='client@example.org/amr'>
  <fields xmlns='urn:xmpp:iot:sensordata' seqnr='1' done='true'>
    <node nodeId='Device01'>
      <timestamp value='2013-03-07T16:24:30'>
        <numeric name='Temperature' momentary='true' automaticReadout=
            'true' value='23.4' unit='°C'/>
      </timestamp>
    </node>
  </fields>
</message>
```

3.2 Read-out failure

If instead a read-out could not be performed, the communication sequence might look as follows:

Listing 4: Momentary read-out failure

```
<iq type='get'
  from='client@example.org/amr'
  to='device@example.org'
  id='S0002'>
  <req xmlns='urn:xmpp:iot:sensordata' seqnr='2' momentary='true'/>
</iq>

<iq type='result'
  from='device@example.org'
  to='client@example.org/amr'
  id='S0002'>
  <accepted xmlns='urn:xmpp:iot:sensordata' seqnr='2'/>
</iq>

<message from='device@example.org'
  to='client@example.org/amr'>
  <failure xmlns='urn:xmpp:iot:sensordata' seqnr='2' done='true'>
```

```
    <error nodeId='Device01' timestamp='2013-03-07T17:13:30'>Timeout.<
        /error>
  </failure>
</message>
```

3.3 Read-out rejected

If for some reason, the device rejects the read-out request, the communication sequence might look as follows:

Listing 5: Momentary read-out rejected

```
<iq type='get'
  from='client@example.org/amr'
  to='device@example.org'
  id='S0003'>
  <req xmlns='urn:xmpp:iot:sensordata' seqnr='3' momentary='true'/>
</iq>

<iq type='error'
  from='device@example.org'
  to='client@example.org/amr'
  id='S0003'>
  <error type='cancel'>
    <forbidden xmlns='urn:ietf:params:xml:ns:xmpp-stanzas'/>
    <text xmlns='urn:ietf:params:xml:ns:xmpp-stanzas' xml:lang='en'>
        Access denied.</text>
  </error>
</iq>
```

Depending on the reason for rejecting the request, different XMPP errors can be returned, according to the description in the following table. The table also lists recommended error type for each error. Any custom error message is returned in a **text** element, as in the example above.

Error Type	Error Element	Namespace	Description
cancel	forbidden	urn:ietf:params:xml:ns:xmpp-stanzas	If the caller lacks privileges to perform the action.
cancel	item-not-found	urn:ietf:params:xml:ns:xmpp-stanzas	If an item or data source could not be found.
modify	bad-request	urn:ietf:params:xml:ns:xmpp-stanzas	If the request was malformed. Examples can include trying to read a device behind a concentrator, without including node information.

3.4 Read-out all

The following example shows a communication sequence when a client reads out all available information from a sensor at a given point in time:

Listing 6: Scheduled read-out of device with multiple responses

```
<iq type='get'
  from='client@example.org/amr'
  to='device@example.org'
  id='S0004'>
  <req xmlns='urn:xmpp:iot:sensordata' seqnr='4' all='true' when='
     2013-03-07T19:00:00'/>
</iq>

<iq type='result'
  from='device@example.org'
  to='client@example.org/amr'
  id='S0004'>
  <accepted xmlns='urn:xmpp:iot:sensordata' seqnr='4' queued='true'/>
</iq>

<message from='device@example.org'
  to='client@example.org/amr'>
  <started xmlns='urn:xmpp:iot:sensordata' seqnr='4'/>
</message>

<message from='device@example.org'
  to='client@example.org/amr'>
  <fields xmlns='urn:xmpp:iot:sensordata' seqnr='4'>
    <node nodeId='Device01'>
      <timestamp value='2013-03-07T19:00:00'>
        <numeric name='Temperature' momentary='true' automaticReadout=
           'true' value='23.4' unit='°C'/>
        <numeric name='Runtime' status='true' automaticReadout='true'
           value='12345' unit='h'/>
        <string name='Device␣ID' identification='true'
           automaticReadout='true' value='Device01'/>
      </timestamp>
    </node>
  </fields>
</message>

<message from='device@example.org'
  to='client@example.org/amr'>
  <fields xmlns='urn:xmpp:iot:sensordata' seqnr='4'>
    <node nodeId='Device01'>
```

```
      <timestamp value='2013-03-07T19:00:00'>
        <numeric name='Temperature, Max' peak='true' automaticReadout=
            'true' value='25.9' unit='°C'/>
        <numeric name='Temperature, Min' peak='true' automaticReadout=
            'true' value='18.7' unit='°C'/>
        <numeric name='Temperature, Mean' computed='true'
            automaticReadout='true' value='22.5' unit='°C'/>
      </timestamp>
    </node>
  </fields>
</message>

<message from='device@example.org'
  to='client@example.org/amr'>
  <fields xmlns='urn:xmpp:iot:sensordata' seqnr='4'>
    <node nodeId='Device01'>
      <timestamp value='2013-03-07T18:00:00'>
        <numeric name='Temperature' historicalHour='true'
            automaticReadout='true' value='24.5' unit='°C'/>
      </timestamp>
      <timestamp value='2013-03-07T17:00:00'>
        <numeric name='Temperature' historicalHour='true'
            automaticReadout='true' value='25.1' unit='°C'/>
      </timestamp>
      <timestamp value='2013-03-07T16:00:00'>
        <numeric name='Temperature' historicalHour='true'
            automaticReadout='true' value='25.2' unit='°C'/>
      </timestamp>
      ...
      <timestamp value='2013-03-07T00:00:00'>
        <numeric name='Temperature' historicalHour='true'
            historicalDay='true' automaticReadout='true' value='25.2'
            unit='°C'/>
      </timestamp>
    </node>
  </fields>
</message>

<message from='device@example.org'
  to='client@example.org/amr'>
  <done xmlns='urn:xmpp:iot:sensordata' seqnr='4'/>
</message>
```

3.5 Read-out of multiple devices

The following example shows how a client reads a subset of multiple sensors behind a device with a single JID.

Listing 7: Read-out of multiple devices

```
<iq type='get'
  from='client@example.org/amr'
  to='device@example.org'
  id='S0005'>
  <req xmlns='urn:xmpp:iot:sensordata' seqnr='5' momentary='true'>
    <node nodeId='Device02'/>
    <node nodeId='Device03'/>
  </req>
</iq>

<iq type='result'
  from='device@example.org'
  to='client@example.org/amr'
  id='S0005'>
  <accepted xmlns='urn:xmpp:iot:sensordata' seqnr='5'/>
</iq>

<message from='device@example.org'
  to='client@example.org/amr'>
  <fields xmlns='urn:xmpp:iot:sensordata' seqnr='5'>
    <node nodeId='Device02'>
      <timestamp value='2013-03-07T19:31:15'>
        <numeric name='Temperature' momentary='true' automaticReadout=
            'true' value='23.4' unit='°C'/>
      </timestamp>
    </node>
  </fields>
</message>

<message from='device@example.org'
  to='client@example.org/amr'>
  <fields xmlns='urn:xmpp:iot:sensordata' seqnr='5' done='true'>
    <node nodeId='Device03'>
      <timestamp value='2013-03-07T19:31:16'>
        <numeric name='Temperature' momentary='true' automaticReadout=
            'true' value='22.8' unit='°C'/>
      </timestamp>
    </node>
  </fields>
</message>
```

3.6 Read-out of specific fields

The **req** element can take **field** sub elements, with which the client can specify which fields it is interested in. If not provided, the client is assumed to return all matching fields, regardless of field name. However, the **field** elements in the request object can be used as a hint which fields should be returned.

Note: the device is not required to adhere to the field limits expressed by these **field** elements. They are considered a hint the device can use to limit bandwidth.
The following example shows how a client can read specific fields in a device.

Listing 8: Read-out of multiple devices

```
<iq type='get'
  from='client@example.org/amr'
  to='device@example.org'
  id='S0006'>
  <req xmlns='urn:xmpp:iot:sensordata' seqnr='6' momentary='true'>
    <node nodeId='Device04'/>
    <field name='Energy'/>
    <field name='Power'/>
  </req>
</iq>

<iq type='result'
  from='device@example.org'
  to='client@example.org/amr'
  id='S0006'>
  <accepted xmlns='urn:xmpp:iot:sensordata' seqnr='6'/>
</iq>

<message from='device@example.org'
  to='client@example.org/amr'>
  <fields xmlns='urn:xmpp:iot:sensordata' seqnr='6' done='true'>
    <node nodeId='Device04'>
      <timestamp value='2013-03-07T22:03:15'>
        <numeric name='Energy' momentary='true' automaticReadout='true
            ' value='12345.67' unit='MWh'/>
        <numeric name='Power' momentary='true' automaticReadout='true'
             value='239.4' unit='W'/>
      </timestamp>
    </node>
  </fields>
</message>
```

3.7 Cancelling a scheduled read-out request

The following example shows how the client cancels a scheduled read-out:

Listing 9: Scheduled read-out of device with multiple responses

```
<iq type='get'
  from='client@example.org/amr'
  to='device@example.org'
  id='S0007'>
```

```
  <req xmlns='urn:xmpp:iot:sensordata' seqnr='8' all='true' when='
     2013-03-09T23:30:00'/>
</iq>

<iq type='result'
  from='device@example.org'
  to='client@example.org/amr'
  id='S0007'>
  <accepted xmlns='urn:xmpp:iot:sensordata' seqnr='8' queued='true'/>
</iq>

<iq type='get'
  from='client@example.org/amr'
  to='device@example.org'
  id='S0008'>
  <cancel xmlns='urn:xmpp:iot:sensordata' seqnr='8'/>
</iq>

<iq type='result'
  from='device@example.org'
  to='client@example.org/amr'
  id='S0008'>
  <cancelled xmlns='urn:xmpp:iot:sensordata' seqnr='8'/>
</iq>
```

4 Determining Support

If an entity supports the protocol specified herein, it MUST advertise that fact by returning a feature of "urn:xmpp:iot:sensordata" in response to Service Discovery (XEP-0030) [2] information requests.

Listing 10: Service discovery information request

```
<iq type='get'
  from='client@example.org/amr'
  to='device@example.org'
  id='disco1'>
  <query xmlns='http://jabber.org/protocol/disco#info'/>
</iq>
```

Listing 11: Service discovery information response

```
<iq type='result'
  from='device@example.org'
  to='client@example.org/amr'
```

[2] XEP-0030: Service Discovery <https://xmpp.org/extensions/xep-0030.html>.

```
  id='disco1'>
  <query xmlns='http://jabber.org/protocol/disco#info'>
    ...
    <feature var='urn:xmpp:iot:sensordata'/>
    ...
  </query>
</iq>
```

In order for an application to determine whether an entity supports this protocol, where possible it SHOULD use the dynamic, presence-based profile of service discovery defined in Entity Capabilities (XEP-0115) [3]. However, if an application has not received entity capabilities information from an entity, it SHOULD use explicit service discovery instead.

5 Implementation Notes

5.1 String lengths

As noticed, a conscious effort has been made not to shorten element and attribute names. This is to make sure, XML is maintained readable. Packet size is not deemed to be affected negatively by this for two reasons:

- For sensors with limited memory, or where package size is important, Efficient XML Interchange (EXI) Format (XEP-0322) [4] is supposed to be used. EXI compresses strings as normalized index values, making the string appear only once in the packet. Therefore, shortening string length doesn't affect packet size much. Element and attribute names in known namespaces are furthermore only encoded by index in schema, not by name.
- If limited memory or package size is not a consideration, readability and ease of implementation is preferred to short messages.

5.2 Enumerations vs. Strings

This protocol has avoided the use of enumerations for data types such as units, field names, etc., and instead use strings. The reasons for this are:

- Enumerations would unnecessarily restrict the use of the protocol to field names and units listed in the protocol.
- It would be very difficult to try to create a complete set of field names and units that would suit all applications.

[3] XEP-0115: Entity Capabilities <https://xmpp.org/extensions/xep-0115.html>.
[4] XEP-0322: Efficient XML Interchange (EXI) Format <https://xmpp.org/extensions/xep-0322.html>.

- Leaving these values as strings would let developers the liberty to use units as they desire.
- If EXI is used for compression, the use of strings will only increase payload slightly, with only one copy of each distinct value used.
- If EXI is not used, this does not affect packet size.

However, some things need to be taken into account:

- Since free strings are used, XML validation cannot be used to secure correct names are used.
- xep-0000-IoT-Interoperability lists recommendations on how field names and units should be used in order to achieve maximum interoperability in SN.
- Consumers of sensor data need to include unit conversion algorithms.

5.3 Asynchronous feedback

Since some applications require real-time feedback (or as real-time as possible), and read-out might in certain cases take a long time, the device has the option to send multiple **fields** messages during read-out. The client is responsible for collecting all such messages until either a **done** message is sent, or a corresponding **done** attribute is available in one of the messages received. Only the device knows how many (if any) messages are sent in response to a read-out request.

5.4 Field Value Data Types

There are different types of values that can be reported from a device. The following table lists the various types:

Element	Description
boolean	Represents a boolean value that can be either true or false.
date	Represents a date value. The value must be encoded using the xs:date data type.
dateTime	Represents a date and optional time value. The value must be encoded using the xs:dateTime data type. This includes date, an optional time and optional time zone information. If time zone is not available, it is supposed to be undefined.
duration	Represents a duration value. The value must be encoded using the xs:duration data type.
enum	Represents an enumeration value. What differs this value from a string value, is that it apart from the enumeration value (which is a string value), also contains a data type, which consumers can use to interpret its value. This specification does not assume knowledge of any particular enumeration data types.

Element	Description
int	Represents a 32-bit integer value. It contains an arbitrary 32-bit integer value. This field value data type can be seen as a subtype of the more generic numeric field value data type. It has its own element, to make it harmonous to 32-bit integer control parameters, as defined in XEP-0325. It is also simpler to report and compress, since it does not use floating point precision and a unit.
long	Represents a 64-bit integer value. It contains an arbitrary 64-bit integer value. This field value data type can be seen as a subtype of the more generic numeric field value data type. It has its own element, to make it harmonous to 64-bit integer control parameters, as defined in XEP-0325. It is also simpler to report and compress, since it does not use floating point precision and a unit.
numeric	Represents a numerical value. Numerical values contain, apart from a numerical number, also an implicit precision (number of decimals) and an optional unit. All parties in the communication chain should retain the number of decimals used, since this contains information that is important in the interpretation of a value. For example, 10 °C is different from 10.0 °C, and very different from 10.00 °C. If a sensor delivers the value 10 °C you can assume it probably lies between 9.5 °C and 10.5 °C. But if a sensor delivers 10.00 °C, it is probably very exact (if calibrated correctly).
string	Represents a string value. It contains an arbitrary string value.
time	Represents a time value. The value must be encoded using the xs:time data type.

5.5 Harmonization with XEP-0325 (Control)

When representing control parameters as momentary field values, it is important to note the similarities and differences between XEP-0323 (this document) and XEP-0325 (Control):
The **enum** field value data type is not available in XEP-0325 (Control). Instead enumeration valued parameters are represented as **string** control parameters, while the control form explicitly lists available options for the parameter. Options are not available in XEP-0323, since it would not be practical to list all options every time the corresponding parameter was read out. Instead, the **enum** element contains a data type attribute, that can be used to identify the type of the enumeration.
The **numeric** field value data type is not available in XEP-0325 (Control). The reason is that a controller is not assumed to understand unit conversion. Any floating-point valued control parameters are represented by **double** control parameters, which lack a unit attribute. They are assumed to have the same unit as the corresponding **numeric** field value. On the other hand, floating point valued control parameters without units, are reported using the **numeric** field element, but leaving the unit blank.
Control pameters of type **color** have no corresponding field value data type. The color value must be represented in another way, and is implementation specific. Possibilities include representing the color as a string, using a specific pattern (for instance RRGGBBAA), or report it using multiple fields, one for each component for instance.

The **boolean**, **date**, **dateTime**, **duration**, **int**, **long**, **string** and **time** field value data types correspond to control parameters having the same types and same element names.

5.6 Field Types

There are different types of fields, apart from types of values a field can have. These types are conceptual types, similar to categories. They are not exclusive, and can be combined.
If requesting multiple field types in a request, the device must interpret this as a union of the corresponding field types and return at least all field values that contain at least one of the requested field types. Example: If requesting momentary values and historical values, devices must return both its momentary values and its historical values.
But, when a device reports a field having multiple field types, the client should interpret this as the intersection of the corresponding field types, i.e. the corresponding field has all corresponding field types. Example: A field marked as both a status value and as a historical value is in fact a historical status value.
The following table lists the different field types specified in this document:

Field Type	Description
computed	A value that is computed instead of measured.
historical*	A value stored in memory from a previous timestamp. The suffix is used to determine period, as shown below.
historicalSecond	A value stored at a second shift (milliseconds = 0).
historicalMinute	A value stored at a minute shift (seconds=milliseconds=0). Are also second values.
historicalHour	A value stored at a hour shift (minutes=seconds=milliseconds=0). Are also minute and second values.
historicalDay	A value stored at a day shift (hours=minutes=seconds=milliseconds=0). Are also hour, minute and second values.
historicalWeek	A value stored at a week shift (Monday, hours=minutes=seconds=milliseconds=0). Are also day, hour, minute and second values.
historicalMonth	A value stored at a month shift (day=1, hours=minutes=seconds=milliseconds=0). Are also day, hour, minute and second values.
historicalQuarter	A value stored at a quarter year shift (Month=Jan, Apr, Jul, Oct, day=1, hours=minutes=seconds=milliseconds=0). Are also month, day, hour, minute and second values.
historicalYear	A value stored at a year shift (Month=Jan, day=1, hours=minutes=seconds=milliseconds=0). Are also quarter, month, day, hour, minute and second values.
historicalOther	If period if historical value is not important in the request or by the device.
identity	A value that can be used for identification. (Serial numbers, meter IDs, locations, names, addresses, etc.)

Field Type	Description
momentary	A momentary value represents a value measured at the time of the read-out. Examples: Energy, Volume, Power, Flow, Temperature, Pressure, etc.
peak	A maximum or minimum value during a given period. Examples ”Temperature, Max”, ”Temperature, Min”, etc.
status	A value displaying status information about something. Examples: Health, Battery life time, Runtime, Expected life time, Signal strength, Signal quality, etc.

There are two field type attributes that can be used in requests to simplify read-out:

Field Type	Description
all	Reads all types of fields. It is the same as explicitly setting all field type attributes to true.
historical	If period of historical values is not important, this attribute can be set to include all types of historical values.

Note: The reason for including different predefined time periods for historical values is that these periods are common in certain applications. However, the client is not restricted to these in any way. The client can always just ask for historical values, and do filtering as necessary to read out the interval desired.
Also, devices are not required to include logic to parse and figure out what historical values are actually desired by the client. If too complicated for the device to handle, it is free to report all historical values. However, the device should limit the historical values to any interval requested, and should try to limit itself to the field types requested. Information in the request element are seen as hints that the device can use to optimize any communication required by the operation.

5.7 Field Quality of Service Values

In applications where quality of service is important, a field must always be accompanied with a corresponding quality of service flag. Devices should set these accordingly. If no quality of service flag is set on a field, the client can assume **automaticReadout** is true.
Note that quality of service flags are not exclusive. Many of them can be logically be combined. Some also imply an order of importance. This should be kept in mind when trying to overwrite existing values with read values: An estimate should not overwrite an automatic read-out, an automatic read-out not a signed value, and a signed value not an invoiced value, etc.
Available quality of service flags, in order of importance:

QoS Flag	Description
missing	Value is missing
inProgress	Value is in progress to be measured or calculated. The value is to be considered as unsure and not final. Read again later to retrieve the correct value. It is more reliable than a missing value, but less reliable than an estimate.
automaticEstimate	An estimate of the value has been done automatically. Considered more reliable than a value in progress.
manualEstimate	The value has manually been estimated. Considered more reliable than an automatic estimate.
manualReadout	Value has been manually read. Considered more reliable than a manual estimate.
automaticReadout	Value has been automatically read. Considered more reliable than a manually read value.
timeOffset	The time was offset more than allowed and corrected during the measurement period.
warning	A warning was logged during the measurement period.
error	An error was logged during the measurement period.
signed	The value has been signed by an operator. Considered more reliable than an automatically read value. Note that the signed quality of service flag can be used to overwrite existing values of higher importance. Example signed + invoiced can be considered more reliable than only invoiced, etc.
invoiced	The value has been invoiced by an operator. Considered more reliable than a signed value.
endOfSeries	The value has been marked as an end point in a series. This can be used for instance to mark the change of tenant in an apartment.
powerFailure	The device recorded a power failure during the measurement period.
invoiceConfirmed	The value has been invoiced by an operator and confirmed by the recipient. Considered more reliable than an invoiced value.

5.7.1 Estimates vs. Readouts

A note on the difference between estimates and readouts. There are many cases where a proper value in a sensor or meter cannot be sensed correctly, and only estimated. As an example: Consider a water meter calculating the flow of water passing vane generating pulses as the wheel turns. The frequency of pulses correspond to the flow of water, or inversely, the time between pulses correspond inversely to the flow of water. But what happens when the flow slows down and pulses are not received? How can the meter differ between zero flow, and a little flow until a pulse is received?
What a meter can do is estimate a flow value that would correspond to the inverse of the time

passed since last received pulse. This estimate would slowly decrease to zero if no flow is available, but would be correct if a pulse finally would be received, thus causing a smoother measurement of the flow. However, the value reported would not be an actual measurement or readout, but an estimate of the value.
It's important that such estimates are flagged as such, so that readers know the value is not a measurement but an estimate. Consider an application that monitors water meters to detect leakage. If a water meter always measures flow, and never decreases to zero flow, it might be logically assumed there's a leakage or bad valves somewhere. However, if meters as described above are used, flow might perhaps never reach zero, simply because it reports a value that's inversely proportional to the time passed since last pulse. It might be close to zero over long periods of time, but never reach zero. To avoid the application generating leakage alarms in case such meters were used, the application can be made to ignore estimates and only monitor values that have been correctly measured.

5.8 Subnodes and supernodes

This document does not go into detail on how devices are ordered behind a JID. Some of the examples have assumed a single device lies behind a JID, others that multiple devices exist behind a JID. Also, no order or structure of devices has been assumed.
But it can be mentioned that it is assumed that if a client requests a read-out of a supernode, it implies the read-out of all its subnodes. Therefore, the client cannot expect read-out to be limited to the devices listed explicitly in a request, as nodes implicitly implied, as descendant nodes of the selected nodes, can also be included.
More information about how multiple devices behind a JID can be handled, is described in the XEP-0326 Internet of Things - Concentrators.

5.9 Reading devices from large subsystems

All examples in this document have been simplified examples where a few devices containing a few fields have been read. However, in many cases large subsystems with very many sensors containing many fields have to be read, as is documented in Internet of Things - Concentrators. In such cases, a node may have to be specified using two or perhaps even three ID's: a **sourceId** identifying the data source controlling the device, a possible **cacheType** narrowing down the search to a specific kind of node, and the common **nodeId**. For more information about this, see Internet of Things - Concentrators.
Note: For cases where the **nodeId** is sufficient to uniquely identify the node, it is sufficient to provide this attribute in the request. If there is ambiguity in the request, the receptor must treat the request as a request with a set of nodes, all with the corresponding **nodeId** as requested.

5.10 Reading controllable parameter values

If reading field values from a device that also supports control through Internet of Things - Control (XEP-0325) [5], the device can report current control parameter values as momentary or status field values, using field names corresponding to its control parameter names. However, such values would probably only correspond to a subset of all data read out. To help the reader to know what fields correspond to controllable parameters, the optional **writable** attribute can be used in responsens. If this attribute is available, it tells the client if the field corresponds to a control parameter with the same name on the device. If the attribute is not available, no deduction can be made if a control parameter with the same name exists or not on the device.

6 Internationalization Considerations

6.1 Time Zones

All timestamps and dateTime values use the XML data type xs:dateTime to specify values. These values include a date, an optional time and an optional time zone.
Note: If time zone is not available, it is supposed to be undefined. The client reading the sensor that reports fields without time zone information should assume the sensor has the same time zone as the client, if not explicitly configured otherwise on the client side.
If devices report time zone, this information should be propagated throughout the system. Otherwise, comparing timestamps from different time zones will be impossible.

6.2 Localized strings

This specification allows for localization of field names in meter data read-out. This is performed by assigning each localizable string a **String ID** which should be unique within a given **Language Module**. A **Language Module** can be any string, including URI's or namespace names. The XEP xep-0000-IoT-Interoperability details how such localizations can be made in an interoperable way.
Note: Localization of strings are for human consumption only. Machines should use the unlocalized strings in program logic.
The following example shows how a device can report localized field information that can be presented to end users without systems being preprogrammed to recognize the device. Language modules can be aggregated by operators after installation, or installed as a pluggable module after the main installation, if localization is desired.

Listing 12: Localized field names

```
<iq type='get'
```

[5] XEP-0325: Internet of Things - Control <https://xmpp.org/extensions/xep-0325.html>.

```
  from='client@example.org/amr'
  to='device@example.org'
  id='S0009'>
  <req xmlns='urn:xmpp:iot:sensordata' seqnr='7' all='true'/>
</iq>

<iq type='result'
  from='device@example.org'
  to='client@example.org/amr'
  id='S0009'>
  <accepted xmlns='urn:xmpp:iot:sensordata' seqnr='7'/>
</iq>

<message from='device@example.org'
  to='client@example.org/amr'>
  <fields xmlns='urn:xmpp:iot:sensordata' seqnr='7' done='true'>
    <node nodeId='Device05'>
      <timestamp value='2013-03-07T22:20:45'>
        <numeric name='Temperature' momentary='true' automaticReadout=
            'true'
          value='23.4' unit='°C' module='Whatchamacallit' stringIds='1
              '/>
        <numeric name='Temperature␣Min' momentary='true'
            automaticReadout='true'
          value='23.4' unit='°C' module='Whatchamacallit' stringIds='
              1,2'/>
        <numeric name='Temperature␣Max' momentary='true'
            automaticReadout='true'
          value='23.4' unit='°C' module='Whatchamacallit' stringIds='
              1,3'/>
        <numeric name='Temperature␣Mean' momentary='true'
            automaticReadout='true'
          value='23.4' unit='°C' module='Whatchamacallit' stringIds='
              1,4'/>
      </timestamp>
    </node>
  </fields>
</message>
```

The above example defines a language module called **Watchamacallit**. In this language module it defines four strings, with IDs 1-4. A system might store these as follows, where the system replaces all **%N%** with a conceptual n:th parameter. (It's up to the system to define these strings, any syntax and how to handle input and output.). In this example, we will assume **%0%** means any previous output, and **%1%** any seed value provided. (See below).

ID	String
1	Temperature

ID	String
2	%0%, Min
3	%0%, Max
4	%0%, Mean

So, when the client reads the field name **Temperature, Min**, it knows that the field name is the composition of the string **Temperature**, and the string **%0%, Min**, where it will replace **%0%** with the output of the previous step, in this case **Temperature**. These strings can later be localized to different languages by operators of the system, and values presented when reading the device, can be done in a language different from the one used by the sensor.
Note: The XEP xep-0000-IoT-Interoperability details how such localizations can be made in an interoperable way.
The **stringIds** attribute merits some further explanation. The value of this attribute must match the following regular expression:

```
^\d+([|]\w+([.]\w+)*([|][^,]*)?)?(,\d+([|]\w+([.]\w+)*([|][^,]*)?)?)*$
```

This basically means, it's of the format: ID_1[|[Module_1][|Seed_1]][...[,ID_n[|[Module_n][|Seed_n]]]]
Where brackets [] mean the contents inside is optional, **ID_i** is an integer representing the string ID in a language module. **Module_i** is optional and allows for specifying a module for **ID_i**, if different from the module defined in the **module** attribute. **Seed_i** allows for seeding the generation of the localized string with a value. This might come in handy when generating strings like **Input 5**, where you don't want to create localized strings for every input there is.
Why such a complicated syntax? The reason is the following: Most localized strings are simple numbers, without the need of specifying modules and seeds. This makes it very efficient to store it as an attribute instead of having to create subelements for every localized field. It's an exception to the rule, to need multiple steps or seeds in the generation of localized strings. Therefore, attributes is an efficient means to specify localization. However, in the general case, a single string ID is not sufficient and multiple steps are required, some seeded.

stringIds	New Parts	Result
1	1="Temperature"	Temperature
1,2	2="%0%, Max"	Temperature, Max
1,1|MathModule	1 in module "MathModule"="sum(%0%)"	sum(Temperature)
3||A1	3="Input %1%"	Input A1
4||A1,2	4="Entrance %1%"	Entrance A1, Max
4||A1,5||3	5="%0%, Floor %1%"	Entrance A1, Floor 3

7 Security Considerations

This document has not touched upon security in sensor networks. There are mainly three concerns that implementers of sensor networks need to consider:

- Communication should be restricted to friends as long as possible. Approved friendships provide a mechanism of limiting sensor information to authorized and authenticated users. However, there are cases where multicast messages may want to go outside of recognized friendships. More information about such use cases, see the XEP xep-0000-IoT-Multicast.
- Sensors may have very limited user interfaces. Even though installation of sensor networks is beyond the scope of this document, a simple installation scheme may include a single LED on the sensor that lights up for a time after receiving a friendship request. If a user presses a button on the device while the LED is lit, the friendship request is acknowledged, and communication is authorized.
- More advanced access rights, privileges, automatic friendship recognition, etc., may be managed by a third party. How to implement more advanced provisioning and detailed access rights to sensor information is detailed in the XEP-0324 Internet of Things - Provisioning. In short, a device, service or user can get a **deviceToken**, **serviceToken** and **userToken** respectivelly from a provisioning server. The service or device then uses these tokens in all readout requests and the device being read out can in turn use these tokens to validate access rights with the provisioning server.

8 IANA Considerations

This document requires no interaction with the Internet Assigned Numbers Authority (IANA) [6].

9 XMPP Registrar Considerations

The protocol schema needs to be added to the list of XMPP protocol schemas.

10 XML Schema

[6] The Internet Assigned Numbers Authority (IANA) is the central coordinator for the assignment of unique parameter values for Internet protocols, such as port numbers and URI schemes. For further information, see <http://www.iana.org/>.

```
<?xml version='1.0' encoding='UTF-8'?>
<xs:schema
  xmlns:xs='http://www.w3.org/2001/XMLSchema'
  targetNamespace='urn:xmpp:iot:sensordata'
  xmlns='urn:xmpp:iot:sensordata'
  elementFormDefault='qualified'>

  <xs:element name='req'>
    <xs:complexType>
      <xs:choice minOccurs='0' maxOccurs='unbounded'>
        <xs:element name='node'>
          <xs:complexType>
            <xs:attribute name='nodeId' type='xs:string' use='required
                '/>
            <xs:attribute name='sourceId' type='xs:string' use='
                optional'/>
            <xs:attribute name='cacheType' type='xs:string' use='
                optional'/>
          </xs:complexType>
        </xs:element>
        <xs:element name='field'>
          <xs:complexType>
            <xs:attribute name='name' type='xs:string' use='required'/
                >
          </xs:complexType>
        </xs:element>
      </xs:choice>
      <xs:attribute name='seqnr' type='xs:int' use='required'/>
      <xs:attributeGroup ref='fieldTypes'/>
      <xs:attribute name='all' type='xs:boolean' use='optional'
          default='false'/>
      <xs:attribute name='historical' type='xs:boolean' use='optional'
           default='false'/>
      <xs:attribute name='from' type='xs:dateTime' use='optional'/>
      <xs:attribute name='to' type='xs:dateTime' use='optional'/>
      <xs:attribute name='when' type='xs:dateTime' use='optional'/>
      <xs:attribute name='serviceToken' type='xs:string' use='optional
          '/>
      <xs:attribute name='deviceToken' type='xs:string' use='optional'
          />
      <xs:attribute name='userToken' type='xs:string' use='optional'/>
    </xs:complexType>
  </xs:element>

  <xs:attributeGroup name='fieldTypes'>
    <xs:attribute name='momentary' type='xs:boolean' use='optional'
        default='false'/>
    <xs:attribute name='peak' type='xs:boolean' use='optional' default
        ='false'/>
```

```
    <xs:attribute name='status' type='xs:boolean' use='optional'
        default='false'/>
    <xs:attribute name='computed' type='xs:boolean' use='optional'
        default='false'/>
    <xs:attribute name='identity' type='xs:boolean' use='optional'
        default='false'/>
    <xs:attribute name='historicalSecond' type='xs:boolean' use='
        optional' default='false'/>
    <xs:attribute name='historicalMinute' type='xs:boolean' use='
        optional' default='false'/>
    <xs:attribute name='historicalHour' type='xs:boolean' use='
        optional' default='false'/>
    <xs:attribute name='historicalDay' type='xs:boolean' use='optional
        ' default='false'/>
    <xs:attribute name='historicalWeek' type='xs:boolean' use='
        optional' default='false'/>
    <xs:attribute name='historicalMonth' type='xs:boolean' use='
        optional' default='false'/>
    <xs:attribute name='historicalQuarter' type='xs:boolean' use='
        optional' default='false'/>
    <xs:attribute name='historicalYear' type='xs:boolean' use='
        optional' default='false'/>
    <xs:attribute name='historicalOther' type='xs:boolean' use='
        optional' default='false'/>
  </xs:attributeGroup>

  <xs:element name='accepted'>
    <xs:complexType>
      <xs:attribute name='seqnr' type='xs:int' use='required'/>
      <xs:attribute name='queued' type='xs:boolean' use='optional'
          default='false'/>
    </xs:complexType>
  </xs:element>

  <xs:element name='started'>
    <xs:complexType>
      <xs:attribute name='seqnr' type='xs:int' use='required'/>
    </xs:complexType>
  </xs:element>

  <xs:element name='cancel'>
    <xs:complexType>
      <xs:attribute name='seqnr' type='xs:int' use='required'/>
    </xs:complexType>
  </xs:element>

  <xs:element name='cancelled'>
    <xs:complexType>
      <xs:attribute name='seqnr' type='xs:int' use='required'/>
```

```
    </xs:complexType>
  </xs:element>

  <xs:element name='fields'>
    <xs:complexType>
      <xs:sequence minOccurs='0' maxOccurs='unbounded'>
        <xs:element name='node'>
          <xs:complexType>
            <xs:sequence minOccurs='0' maxOccurs='unbounded'>
              <xs:element name='timestamp'>
                <xs:complexType>
                  <xs:choice minOccurs='0' maxOccurs='unbounded'>
                    <xs:element name='boolean' type='boolean'/>
                    <xs:element name='date' type='date'/>
                    <xs:element name='dateTime' type='dateTime'/>
                    <xs:element name='duration' type='duration'/>
                    <xs:element name='enum' type='enum'/>
                    <xs:element name='int' type='int'/>
                    <xs:element name='long' type='long'/>
                    <xs:element name='numeric' type='numeric'/>
                    <xs:element name='string' type='string'/>
                    <xs:element name='time' type='time'/>
                  </xs:choice>
                  <xs:attribute name='value' type='xs:dateTime' use='
                      required'/>
                </xs:complexType>
              </xs:element>
            </xs:sequence>
            <xs:attribute name='nodeId' type='xs:string' use='required
                '/>
            <xs:attribute name='sourceId' type='xs:string' use='
                optional'/>
            <xs:attribute name='cacheType' type='xs:string' use='
                optional'/>
          </xs:complexType>
        </xs:element>
      </xs:sequence>
      <xs:attribute name='seqnr' type='xs:int' use='required'/>
      <xs:attribute name='done' type='xs:boolean' use='optional'
          default='false'/>
    </xs:complexType>
  </xs:element>

  <xs:element name='failure'>
    <xs:complexType>
      <xs:sequence minOccurs='0' maxOccurs='unbounded'>
        <xs:element name='error'>
          <xs:complexType>
            <xs:simpleContent>
```

```
            <xs:extension base='xs:string'>
              <xs:attribute name='nodeId' type='xs:string' use='
                  required'/>
              <xs:attribute name='sourceId' type='xs:string' use='
                  optional'/>
              <xs:attribute name='cacheType' type='xs:string' use='
                  optional'/>
              <xs:attribute name='timestamp' type='xs:string' use='
                  required'/>
            </xs:extension>
          </xs:simpleContent>
        </xs:complexType>
      </xs:element>
    </xs:sequence>
    <xs:attribute name='seqnr' type='xs:int' use='required'/>
    <xs:attribute name='done' type='xs:boolean' use='optional'
       default='false'/>
  </xs:complexType>
</xs:element>

<xs:element name='done'>
  <xs:complexType>
    <xs:attribute name='seqnr' type='xs:int' use='required'/>
  </xs:complexType>
</xs:element>

<xs:complexType name='field' abstract='true'>
  <xs:attribute name="name" type="xs:string" use="required"/>
  <xs:attributeGroup ref='fieldTypes'/>
  <xs:attributeGroup ref='fieldQoS'/>
  <xs:attribute name="module" type="xs:string" use="optional"/>
  <xs:attribute name="stringIds" type="StringIds" use="optional"/>
  <xs:attribute name="writable" type="xs:boolean" use="optional"/>
</xs:complexType>

<xs:simpleType name="StringIds">
  <xs:restriction base="xs:string">
    <xs:pattern value="^\d+([|]\w+([.]\w+)*([|][^,]*)?)?(,\d+([|]\w
       +([.]\w+)*([|][^,]*)?)?)*$"/>
  </xs:restriction>
</xs:simpleType>

<xs:attributeGroup name='fieldQoS'>
  <xs:attribute name='missing' type='xs:boolean' use='optional'
     default='false'/>
  <xs:attribute name='inProgress' type='xs:boolean' use='optional'
     default='false'/>
  <xs:attribute name='automaticEstimate' type='xs:boolean' use='
     optional' default='false'/>
```

```
    <xs:attribute name='manualEstimate' type='xs:boolean' use='
        optional' default='false'/>
    <xs:attribute name='manualReadout' type='xs:boolean' use='optional
        ' default='false'/>
    <xs:attribute name='automaticReadout' type='xs:boolean' use='
        optional' default='false'/>
    <xs:attribute name='timeOffset' type='xs:boolean' use='optional'
        default='false'/>
    <xs:attribute name='warning' type='xs:boolean' use='optional'
        default='false'/>
    <xs:attribute name='error' type='xs:boolean' use='optional'
        default='false'/>
    <xs:attribute name='signed' type='xs:boolean' use='optional'
        default='false'/>
    <xs:attribute name='invoiced' type='xs:boolean' use='optional'
        default='false'/>
    <xs:attribute name='endOfSeries' type='xs:boolean' use='optional'
        default='false'/>
    <xs:attribute name='powerFailure' type='xs:boolean' use='optional'
         default='false'/>
    <xs:attribute name='invoiceConfirmed' type='xs:boolean' use='
        optional' default='false'/>
  </xs:attributeGroup>

  <xs:complexType name='numeric'>
    <xs:complexContent>
      <xs:extension base='field'>
        <xs:attribute name="value" type="xs:double" use="required"/>
        <xs:attribute name="unit" type="xs:string" use="required"/>
      </xs:extension>
    </xs:complexContent>
  </xs:complexType>

  <xs:complexType name='string'>
    <xs:complexContent>
      <xs:extension base='field'>
        <xs:attribute name="value" type="xs:string" use="required"/>
      </xs:extension>
    </xs:complexContent>
  </xs:complexType>

  <xs:complexType name='boolean'>
    <xs:complexContent>
      <xs:extension base='field'>
        <xs:attribute name="value" type="xs:boolean" use="required"/>
      </xs:extension>
    </xs:complexContent>
  </xs:complexType>
```

```
<xs:complexType name='date'>
  <xs:complexContent>
    <xs:extension base='field'>
      <xs:attribute name="value" type="xs:date" use="required"/>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>

<xs:complexType name='dateTime'>
  <xs:complexContent>
    <xs:extension base='field'>
      <xs:attribute name="value" type="xs:dateTime" use="required"/>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>

<xs:complexType name='duration'>
  <xs:complexContent>
    <xs:extension base='field'>
      <xs:attribute name="value" type="xs:duration" use="required"/>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>

<xs:complexType name='enum'>
  <xs:complexContent>
    <xs:extension base='field'>
      <xs:attribute name="value" type="xs:string" use="required"/>
      <xs:attribute name="dataType" type="xs:string" use="required"/
          >
    </xs:extension>
  </xs:complexContent>
</xs:complexType>

<xs:complexType name='int'>
  <xs:complexContent>
    <xs:extension base='field'>
      <xs:attribute name="value" type="xs:int" use="required"/>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>

<xs:complexType name='long'>
  <xs:complexContent>
    <xs:extension base='field'>
      <xs:attribute name="value" type="xs:long" use="required"/>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>
```

```
  <xs:complexType name='time'>
    <xs:complexContent>
      <xs:extension base='field'>
        <xs:attribute name="value" type="xs:time" use="required"/>
      </xs:extension>
    </xs:complexContent>
  </xs:complexType>

</xs:schema>
```

11 For more information

For more information, please see the following resources:

- The Sensor Network section of the XMPP Wiki contains further information about the use of the sensor network XEPs, links to implementations, discussions, etc.
- The XEP's and related projects are also available on github, thanks to Joachim Lindborg.
- A presentation giving an overview of all extensions related to Internet of Things can be found here: http://prezi.com/esosntqhewhs/iot-xmpp/.

12 Acknowledgements

Thanks to Flemon Ghobrial, Joachim Lindborg, Karin Forsell, Tina Beckman, Kevin Smith and Tobias Markmann for all valuable feedback.