XEP-0326: Internet of Things - Concentrators

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

2017-05-20
Version 0.4

Status	Type	Short Name
Retracted	Standards Track	sensor-network-concentrators

Note: This specification has been retracted by the author; new implementations are not recommended. This specification describes how to manage and get information from concentrators of devices over XMPP networks.

Legal

Copyright

Permissions

Warranty

Liability

Conformance

Contents

1 Introduction

Concentrators are devices in sensor networks, concentrating the management of a sub set of devices to one point. They can be small (for example: PLC:s managing a small set of sensors and actuators), medium-sized (for example: mid-level concentrators, controlling branches of the network, islands, perhaps using separate communication protocols), large (for example: entire sub-systems, perhaps managed by a separate child/partner organization) to massive (for example: The entire top-level system, smart-grid, IoT network).
Even though this XEP is generally written and can be used by other implementations not based on sensor networks, much of the requirements used to define this specification comes from requirements used in sensor networks and Internet of Things applications and infrastructure. This specification will define the following aspects of a general concentrator profile, that can handle all different types of concentrators available in sensor network architectures:

- A concentrator works with multiple **data sources**. Effective management of data sources and their contents is a vital part of this XEP.
- The ability to work with massive quantities of entities.
- Effective synchronization of contents between interested parties.
- Effective ways to interact with entities controlled by the concentrator.

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-Discovery	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.
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	Description
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	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-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	This specification. 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>..

1.1 Relations to other extensions

Even though there are technologies available in forms of XEPs that solve parts of the above mentioned problem, they do not provide sufficient support. The following paragraphs will take the time to list why different technologies are not applicable.

1.1.1 XEP-0060

This XEP defines tree structures for nodes in different data sources. Publish-Subscribe (XEP-0060) [1] defines a model where a tree structure of nodes is published and users can browse this tree structure. Furthermore, it allows the possibility to publish items on these nodes as well as syndication of this information.
This XEP also defines data sources (in a tree structure). These data sources contain nodes. PubSub Collection Nodes (XEP-0248) [2] defines a structure called a node collection, a structure that allows the creation of collections containing loosely coupled nodes.
Even though this document defines tree structures of data, it is not however based on XEP-0060. There are multiple reasons for this:

- The structures defined in this specification do not include items to publish for each node.
- We want to be able to use XEP-0060 in parallel to this specification, for the purpose of publishing sensor data. More information about this is found in xep-0000-IoT-PubSub.html.
- For massive systems (hundreds of thousands, or millions, of nodes behind a concentrator, its vitally important to be able to manage sets of nodes directly (for example: Edit multiple nodes at once). Many of the operations in XEP-0060 only allow for operations of singular nodes. Furthermore, many simple operations require multiple messages per node. This document defines way to operate of sets of nodes simultaneously, as well as ways to perform operations with a smaller number of operations.
- In this document, nodes have specific functions, controlled by a specific Node Type. Different Node Types have different parameter sets, different options, commands, capabilities, etc. XEP-0060 does not differ between node types. There, nodes are only a structural way to sort data into a tree graph.
- In this document, nodes have real-time status, like errors, warnings, etc.

1.1.2 XEP-0248

XEP-0248 defines the concept of node collections and syndication of information from nodes in these collections. But XEP-0248 is not used in this specification. There are multiple reasons:

[1] XEP-0060: Publish-Subscribe <https://xmpp.org/extensions/xep-0060.html>.
[2] XEP-0248: PubSub Collection Nodes <https://xmpp.org/extensions/xep-0248.html>.

- We want to be able to use XEP-0248 in parallel to this specification, for the purpose of publishing sensor data. More information about this is found in xep-0000-IoT-PubSub.html.
- Node IDs are not necessarily unique by themselves in the system. This document defines a uniqueness concept based on a triple of data: (Data Source ID, Cache Type, Node ID). This means that Nodes must have IDs unique within a given Cache Type, within a given data source.
- We need to expand on types of events generated from a data source, to make them adhere to the particulars of nodes as defined in this specification.
- Data sources own their nodes. XEP-0248 define a loosely coupled structure with references to nodes. In this document, a data source is the owner of all nodes contained in it.

1.1.3 XEP-0050

Ad-Hoc Commands (XEP-0050) [3] defines how ad-hoc commands can be implemented and how clients can use such commands to interact with underlying logic. But XEP-0050 is not used in this specification. There are multiple reasons:

- We want to be able to use XEP-0050 for other types of commands, than commands defined in this specification. Generally, XEP-0050 is used to implement system-wide commands.
- Commands defined in this specification are context sensitive, i.e. they depend on the type of node and the context of the node on which the act.
- It is a requirement to be able to execute commands on sets of nodes directly.
- Since commands have to be context sensitive, a large concentrator system may have hundreds or thousands of different commands, making it impossible to create context sensitive GUI's using XEP-0050.
- Dialog types used for Ad-Hoc-commands are not sufficient. First, dynamic dialogs are required in the general case. (XEP-0326 define how to create dynamic forms.) Furthermore, the wizard style type of dialogs used for more complex dialogs in ad-hoc commands, are difficult to automate.

2 Glossary

The following table lists common terms and corresponding descriptions.

[3] XEP-0050: Ad-Hoc Commands <https://xmpp.org/extensions/xep-0050.html>.

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.

Data Source A Data source contains a collection of nodes. Three types of data sources exist: Singular, Flat and Tree. Singular data sources only include one object. Flat data sources contain a list of objects and Tree data sources contain nodes formed as a tree graph with one root element.

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 readout.

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. Nodes belong to a data source, and all nodes have a Node Type. Some nodes have a parent node, and some nodes have child nodes. Nodes with the same parent nodes a called sibling nodes.

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.

Node Type Each node has a Node Type. The Node Type defines the functionality of the node in the system.

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

To create a complete set of operations supported by all types of concentrators, ranging from PLCs to subsystems to entire systems is very difficult. So, the aim of this document is instead to create a very small reduced set of operations, a common denominator, that would allow for

basic maintenance and interoperability of concentrators of different makes and models and of these varying ranges.

3.1 Capabilities

3.1.1 Get Capabilities

This document lists a sequence of commands. Some are very basic, while others are used for managing massive amounts of devices. When developing a small PLC, it might be difficult to motivate the implementation of the more advanced commands. They are simply not necessary for the management of the device. So, clients connecting to the concentrator need a way to learn what operations are available in the concentrator, and as a consequence what operations are not. To do this, the **getCapabilities** command is sent, as is shown in the following example.

Listing 1: Full capabilities

```
<iq type='get'
  from='client@example.org/client'
  to='subsystem@example.org'
  id='1'>
  <getCapabilities xmlns='urn:xmpp:iot:concentrators'/>
</iq>

<iq type='result'
  from='subsystem@example.org'
  to='client@example.org/client'
  id='1'>
  <getCapabilitiesResponse xmlns='urn:xmpp:iot:concentrators' result='
      OK'>
    <value>getCapabilities</value>
    <value>getAllDataSources</value>
    <value>getRootDataSources</value>
    <value>getChildDataSources</value>
    <value>containsNode</value>
    <value>containsNodes</value>
    <value>getNode</value>
    <value>getNodes</value>
    <value>getAllNodes</value>
    <value>getNodeInheritance</value>
    <value>getRootNodes</value>
    <value>getChildNodes</value>
    <value>getIndices</value>
    <value>getNodesFromIndex</value>
    <value>getNodesFromIndices</value>
    <value>getAllIndexValues</value>
    <value>getNodeParametersForEdit</value>
    <value>setNodeParametersAfterEdit</value>
```

```
    <value>getCommonNodeParametersForEdit</value>
    <value>setCommonNodeParametersAfterEdit</value>
    <value>getAddableNodeTypes</value>
    <value>getParametersForNewNode</value>
    <value>createNewNode</value>
    <value>destroyNode</value>
    <value>getAncestors</value>
    <value>getNodeCommands</value>
    <value>getCommandParameters</value>
    <value>executeNodeCommand</value>
    <value>executeNodeQuery</value>
    <value>abortNodeQuery</value>
    <value>getCommonNodeCommands</value>
    <value>getCommonCommandParameters</value>
    <value>executeCommonNodeCommand</value>
    <value>executeCommonNodeQuery</value>
    <value>abortCommonNodeQuery</value>
    <value>moveNodeUp</value>
    <value>moveNodeDown</value>
    <value>moveNodesUp</value>
    <value>moveNodesDown</value>
    <value>subscribe</value>
    <value>unsubscribe</value>
    <value>getDatabases</value>
    <value>getDatabaseReadoutParameters</value>
    <value>startDatabaseReadout</value>
  </getCapabilitiesResponse>
</iq>
```

A concentrator without databases, but still contain a rich interface for handling masses of nodes may present itself as follows:

Listing 2: No database capabilities

```
<iq type='get'
  from='client@example.org/client'
  to='subsystem@example.org'
  id='63'>
  <getCapabilities xmlns='urn:xmpp:iot:concentrators'/>
</iq>

<iq type='result'
  from='subsystem@example.org'
  to='client@example.org/client'
  id='63'>
  <getCapabilitiesResponse xmlns='urn:xmpp:iot:concentrators' result='
      OK'>
    <value>getCapabilities</value>
    <value>getAllDataSources</value>
```

```
    <value>getRootDataSources</value>
    <value>getChildDataSources</value>
    <value>containsNode</value>
    <value>containsNodes</value>
    <value>getNode</value>
    <value>getNodes</value>
    <value>getAllNodes</value>
    <value>getNodeInheritance</value>
    <value>getRootNodes</value>
    <value>getChildNodes</value>
    <value>getIndices</value>
    <value>getNodesFromIndex</value>
    <value>getNodesFromIndices</value>
    <value>getAllIndexValues</value>
    <value>getNodeParametersForEdit</value>
    <value>setNodeParametersAfterEdit</value>
    <value>getCommonNodeParametersForEdit</value>
    <value>setCommonNodeParametersAfterEdit</value>
    <value>getAddableNodeTypes</value>
    <value>getParametersForNewNode</value>
    <value>createNewNode</value>
    <value>destroyNode</value>
    <value>getAncestors</value>
    <value>getNodeCommands</value>
    <value>getCommandParameters</value>
    <value>executeNodeCommand</value>
    <value>executeNodeQuery</value>
    <value>abortNodeQuery</value>
    <value>getCommonNodeCommands</value>
    <value>getCommonCommandParameters</value>
    <value>executeCommonNodeCommand</value>
    <value>executeCommonNodeQuery</value>
    <value>abortCommonNodeQuery</value>
    <value>moveNodeUp</value>
    <value>moveNodeDown</value>
    <value>moveNodesUp</value>
    <value>moveNodesDown</value>
    <value>subscribe</value>
    <value>unsubscribe</value>
  </getCapabilitiesResponse>
</iq>
```

A smaller gateway on the other hand, may have skipped the implementation of the batch commands that are used for larger systems:

Listing 3: No batch command capabilities

```
<iq type='get'
  from='client@example.org/client'
```

```
  to='gateway@example.org'
  id='2'>
  <getCapabilities xmlns='urn:xmpp:iot:concentrators'/>
</iq>

<iq type='result'
  from='gateway@example.org'
  to='client@example.org/client'
  id='2'>
  <getCapabilitiesResponse xmlns='urn:xmpp:iot:concentrators' result='
      OK'>
    <value>getCapabilities</value>
    <value>getAllDataSources</value>
    <value>getRootDataSources</value>
    <value>getChildDataSources</value>
    <value>containsNode</value>
    <value>getNode</value>
    <value>getNodeInheritance</value>
    <value>getRootNodes</value>
    <value>getChildNodes</value>
    <value>getNodeParametersForEdit</value>
    <value>setNodeParametersAfterEdit</value>
    <value>getAddableNodeTypes</value>
    <value>getParametersForNewNode</value>
    <value>createNewNode</value>
    <value>destroyNode</value>
    <value>getAncestors</value>
    <value>getNodeCommands</value>
    <value>getCommandParameters</value>
    <value>executeNodeCommand</value>
    <value>executeNodeQuery</value>
    <value>abortNodeQuery</value>
    <value>moveNodeUp</value>
    <value>moveNodeDown</value>
    <value>moveNodesUp</value>
    <value>moveNodesDown</value>
    <value>subscribe</value>
    <value>unsubscribe</value>
  </getCapabilitiesResponse>
</iq>
```

But a small PLC, possibly with a fixed set of nodes, might have support for an even more reduced set of commands:

Listing 4: No edit capabilities

```
<iq type='get'
  from='client@example.org/client'
  to='plc@example.org'
```

```
  id='3'>
  <getCapabilities xmlns='urn:xmpp:iot:concentrators'/>
</iq>

<iq type='result'
  from='plc@example.org'
  to='client@example.org/client'
  id='3'>
  <getCapabilitiesResponse xmlns='urn:xmpp:iot:concentrators' result='
      OK'>
    <value>getCapabilities</value>
    <value>getAllDataSources</value>
    <value>containsNode</value>
    <value>getNode</value>
    <value>getRootNodes</value>
    <value>getChildNodes</value>
    <value>getNodeCommands</value>
    <value>getCommandParameters</value>
    <value>executeNodeCommand</value>
    <value>executeNodeQuery</value>
    <value>abortNodeQuery</value>
  </getCapabilitiesResponse>
</iq>
```

So, clients who need to interact with different types of concentrators need to be aware of what commands are supported, and limit operations to those commands.

3.2 Data Sources

3.2.1 Get All Data Sources

This command will return a flat list of all available data sources on the concentrator. It is not structured hierarchically.

Listing 5: Get All Data Sources

```
<iq type='get'
  from='client@example.org/client'
  to='concentrator@example.org'
  id='4'>
  <getAllDataSources xmlns='urn:xmpp:iot:concentrators' xml:lang='en'/
      >
</iq>

<iq type='result'
  from='concentrator@example.org'
  to='client@example.org/client'
  id='4'>
```

```
  <getAllDataSourcesResponse xmlns='urn:xmpp:iot:concentrators' result
     ='OK'>
    <dataSource sourceId='Applications' name='Applications'
       hasChildren='false' lastChanged='2013-03-19T17:58:01'/>
    <dataSource sourceId='Certificates' name='Certificates'
       hasChildren='false' lastChanged='2013-02-20T12:31:54'/>
    <dataSource sourceId='Clayster.EventSink.Programmable' name='
       Programmable␣Event␣Log' hasChildren='false' lastChanged='
       2012-10-25T09:31:12'/>
    ...
  </getAllDataSourcesResponse>
</iq>
```

3.2.2 Get Root Data Sources

If the client is interested in the hierarchical structure of available data sources, it should request only the root sources, and then ask the client for their corresponding child data sources. If the client wants to present the data sources to a user, presenting them in their hierarchical order may be more intuitive.

Listing 6: Get Root Data Sources

```
<iq type='get'
  from='client@example.org/client'
  to='concentrator@example.org'
  id='5'>
  <getRootDataSources xmlns='urn:xmpp:iot:concentrators' xml:lang='en'
     />
</iq>

<iq type='result'
  from='concentrator@example.org'
  to='client@example.org/client'
  id='5'>
  <getRootDataSourcesResponse xmlns='urn:xmpp:iot:concentrators'
     result='OK'>
    <dataSource sourceId='MeteringRoot' name='Metering' hasChildren='
       true' lastChanged='2013-03-19T17:58:01'/>
    <dataSource sourceId='SecurityRoot' name='Security' hasChildren='
       true' lastChanged='2013-01-12T22:03:50'/>
    <dataSource sourceId='SystemRoot' name='System' hasChildren='true'
        lastChanged='2012-02-20T12:34:56'/>
    ...
  </getRootDataSourcesResponse>
</iq>
```

3.2.3 Get Child Data Sources

Having the ID of a data source that contains child data sources, you can fetch the child sources as follows:

Listing 7: Get Child Data Sources

```
<iq type='get'
  from='client@example.org/client'
  to='concentrator@example.org'
  id='6'>
  <getChildDataSources xmlns='urn:xmpp:iot:concentrators' sourceId='
     MeteringRoot' xml:lang='en'/>
</iq>

<iq type='result'
  from='concentrator@example.org'
  to='client@example.org/client'
  id='6'>
  <getChildDataSourcesResponse xmlns='urn:xmpp:iot:concentrators'
     result='OK'>
    <dataSource sourceId='MeteringFieldImports' name='Field Imports'
       hasChildren='false' lastChanged='2013-03-19T17:58:01'/>
    <dataSource sourceId='MeteringFieldProcessors' name='Field
       Processors' hasChildren='false' lastChanged='2013-03-19
       T17:58:01'/>
    <dataSource sourceId='MeteringFieldSinks' name='Field Sinks'
       hasChildren='false' lastChanged='2013-03-19T17:58:01'/>
    <dataSource sourceId='MeteringGroups' name='Groups' hasChildren='
       false' lastChanged='2013-03-19T17:58:01'/>
    <dataSource sourceId='MeteringJobs' name='Jobs' hasChildren='false
       ' lastChanged='2013-03-19T17:58:01'/>
    <dataSource sourceId='MeteringTopology' name='Topology'
       hasChildren='false' lastChanged='2013-03-19T17:58:01'/>
    <dataSource sourceId='MeteringUnitConversion' name='Unit
       Conversion' hasChildren='false' lastChanged='2013-03-19
       T17:58:01'/>
  </getChildDataSourcesResponse>
</iq>
```

3.2.4 Subscribe to data source events

A client can subscribe to changes made in a data source. It does this by sending the **subscribe** command to the concentrator, as is shown in the following example:

Listing 8: Subscribing to data source events

```
<iq type='set'
```

```
  from='client@example.org/client'
  to='concentrator@example.org'
  id='57'>
  <subscribe xmlns='urn:xmpp:iot:concentrators' sourceId='
      MeteringTopology'/>
</iq>

<iq type='result'
  from='concentrator@example.org'
  to='client@example.org/client'
  id='57'>
  <subscribeResponse xmlns='urn:xmpp:iot:concentrators' result='OK'/>
</iq>
```

Multiple subscriptions to the same source will not result in an error, however the server will still only send one event message for each event in the data source.

Important: Event subscriptions only last for as long as the client and concentrator both maintain presence. The concentrator must not persist event notification subscriptions, and if it goes offline and back online, or if the client goes offline or online again for any reason, the event subscription is removed.

Note: The **parameters** and **messages** attributes can be used to retrieve parameter and status message information about the nodes in event messages sent from the concentrator. Note that the **xml:lang** may be used to select the language used in such events, if the concentrator supports localization of strings.

Listing 9: Subscribing to data source events with localized parameters

```
<iq type='set'
  from='client@example.org/client'
  to='concentrator@example.org'
  id='60'>
  <subscribe xmlns='urn:xmpp:iot:concentrators' sourceId='
      MeteringTopology' parameters='true' messages='true' xml:lang='en
      '/>
</iq>

<iq type='result'
  from='concentrator@example.org'
  to='client@example.org/client'
  id='60'>
  <subscribeResponse xmlns='urn:xmpp:iot:concentrators' result='OK'/>
</iq>
```

The **subscribe** command has a set of optional attributes, one for each event type available, and with the same names (**nodeAdded**, **nodeUpdated**, **nodeStatusChanged**, **nodeRemoved**, **nodeMovedUp** and **nodeMovedDown**), that the client can use to subscribe to individual events, but not to others. They have the default value of true implying that if not provided, the default action is to subscribe to those events. The attributes **parameters** and **messages**

can also be used to specify if node parameters and node messages respectively should be available in event messages. The default value for the these later attributes is false, implying that normal events do not include node parameter and node message information.
The following example shows how a client can subscribe to a set of events only:

Listing 10: Subscribing to data source events, avoiding state events

```
<iq type='set'
  from='client@example.org/client'
  to='concentrator@example.org'
  id='61'>
  <subscribe xmlns='urn:xmpp:iot:concentrators' sourceId='
      MeteringTopology' parameters='true' messages='false' xml:lang='
      en'
    nodeAdded='true' nodeUpdated='true' nodeStatusChanged='false'
       nodeRemoved='true' nodeMovedUp='false' nodeMovedDown='false'/>
</iq>

<iq type='result'
  from='concentrator@example.org'
  to='client@example.org/client'
  id='61'>
  <subscribeResponse xmlns='urn:xmpp:iot:concentrators' result='OK'/>
</iq>
```

For more information on types of events sent, see the Data Source Events section.

3.2.5 Unsubscribe from data source events

A client can unsubscribe to changes made in a data source it is subscribed to. It does this by sending the **unsubscribe** command to the concentrator, as is shown in the following example:

Listing 11: Unsubscribing from data source events

```
<iq type='set'
  from='client@example.org/client'
  to='concentrator@example.org'
  id='58'>
  <unsubscribe xmlns='urn:xmpp:iot:concentrators' sourceId='
      MeteringTopology'/>
</iq>

<iq type='result'
  from='concentrator@example.org'
  to='client@example.org/client'
  id='58'>
```

```
  <unsubscribeResponse xmlns='urn:xmpp:iot:concentrators' result='OK'/
      >
</iq>
```

An unsubscription made to an existing data source, but where an event subscription does not exist, must not result in an error.
The **unsubscribe** command has a set of optional attributes, one for each event type available, and with the same names, that the client can use to unsubscribe from individual events, but not from others. They have the default value of true implying that if not provided, the default action is to unsubscribe from those events.
The following example shows how a client can unsubscribe from a subset of events, keeping subscriptions on the others (if subscribed to):

Listing 12: Unsubscribing from state events

```
<iq type='set'
  from='client@example.org/client'
  to='concentrator@example.org'
  id='62'>
  <unsubscribe xmlns='urn:xmpp:iot:concentrators' sourceId='
      MeteringTopology' parameters='true' messages='false' xml:lang='
      en'
    nodeAdded='false' nodeUpdated='false' nodeStatusChanged='true'
       nodeRemoved='false' nodeMovedUp='false' nodeMovedDown='false'/
       >
</iq>

<iq type='result'
  from='concentrator@example.org'
  to='client@example.org/client'
  id='62'>
  <subscribeResponse xmlns='urn:xmpp:iot:concentrators' result='OK'/>
</iq>
```

3.2.6 Get changes since given timestamp before subscribing

If a client comes back online and wants to know any changes that have taken place on the concentrator since last time it was in contact with it, it can include a **getEventsSince** attribute in the **subscribe** command sent to the concentrator. This will make the concentrator send all event messages since the given timestamp to the client before subscribing the client to events in the given data source.

Listing 13: Get changes since given timestamp before subscribing

```
<iq type='set'
  from='client@example.org/client'
  to='concentrator@example.org'
```

```
  id='59'>
  <subscribe xmlns='urn:xmpp:iot:concentrators' sourceId='
      MeteringTopology' getEventsSince='2013-03-21T19:24:00'/>
</iq>

<iq type='result'
  from='concentrator@example.org'
  to='client@example.org/client'
  id='59'>
  <subscribeResponse xmlns='urn:xmpp:iot:concentrators' result='OK'/>
</iq>

... Sequence of event messages sent from concentrator to client.
```

Important: Event subscriptions only last for as long as the client and concentrator both maintain presence. The concentrator must not persist event notification subscriptions, and if it goes offline and back online, or if the client goes offline or online again for any reason, the event subscription is removed.
Note: The **parameters** and **messages** attributes can be used to retrieve parameter and status message information about the nodes in event messages sent from the concentrator.
For more information on types of events sent, see the Data Source Events section.

3.2.7 Get changes since given timestamp before subscribing, Failure

If during a subscription request the concentrator is not able to fulfill the request of retrieving previous events using the **getEventsSince** attribute, perhaps the attribute stretches too far back, or includes too many records, the concentrator can return an error message using a response code of **NotImplemented**. In this case, the subscription must not be made.
When receiving such an error from the concentrator, the client must make a decision if it should download the data source again, or keep the data source as is, and subscribing again without the **getEventsSince** attribute.

Listing 14: Get changes since given timestamp before subscribing, Failure

```
<iq type='set'
  from='client@example.org/client'
  to='concentrator@example.org'
  id='73'>
  <subscribe xmlns='urn:xmpp:iot:concentrators' sourceId='
      MeteringTopology' getEventsSince='2001-01-01T00:00:00'/>
</iq>

<iq type='result'
  from='concentrator@example.org'
  to='client@example.org/client'
  id='73'>
```

```
  <subscribeResponse xmlns='urn:xmpp:iot:concentrators' result='
      NotImplemented'/>
</iq>
```

3.3 Nodes

3.3.1 Contains Node

This command permits the client to check the existence of a node in the concentrator.

Listing 15: Checking the existence of a node

```
<iq type='get'
  from='client@example.org/client'
  to='concentrator@example.org'
  id='7'>
  <containsNode xmlns='urn:xmpp:iot:concentrators' sourceId='
      MeteringTopology' nodeId='Node1'/>
</iq>

<iq type='result'
  from='concentrator@example.org'
  to='client@example.org/client'
  id='7'>
  <containsNodeResponse xmlns='urn:xmpp:iot:concentrators' result='OK'
      >true</containsNodeResponse>
</iq>
```

3.3.2 Contains Nodes

If the client wants to check the existence of multiple nodes on the concentrator, it can use this batch command instead:

Listing 16: Checking the existence of a multiple nodes

```
<iq type='get'
  from='client@example.org/client'
  to='concentrator@example.org'
  id='8'>
  <containsNodes xmlns='urn:xmpp:iot:concentrators'>
    <node sourceId='MeteringTopology' nodeId='Node1'/>
    <node sourceId='MeteringTopology' nodeId='Node2'/>
    <node sourceId='MeteringTopology' nodeId='Node3'/>
    <node sourceId='MeteringGroups' nodeId='Group1'/>
  </containsNodes>
</iq>
```

```
<iq type='result'
  from='concentrator@example.org'
  to='client@example.org/client'
  id='8'>
  <containsNodesResponse xmlns='urn:xmpp:iot:concentrators' result='OK
     '>
    <value>true</value>
    <value>true</value>
    <value>false</value>
    <value>true</value>
  </containsNodesResponse>
</iq>
```

The array returned will have one item for each item in the request, in the same order.

3.3.3 Get Node

This command returns basic information about a node in the concentrator.

Listing 17: Get Node

```
<iq type='get'
  from='client@example.org/client'
  to='concentrator@example.org'
  id='9'>
  <getNode xmlns='urn:xmpp:iot:concentrators' sourceId='
     MeteringTopology' nodeId='Node1' xml:lang='en'/>
</iq>

<iq type='result'
  from='concentrator@example.org'
  to='client@example.org/client'
  id='9'>
  <getNodeResponse xmlns='urn:xmpp:iot:concentrators'
    result='OK'
    nodeId='Node1'
    nodeType='Namespace.NodeType1'
    cacheType='Node'
    state='WarningUnsigned'
    hasChildren='false'
    isReadable='true'
    isControllable='true'
    hasCommands='true'
    parentId='Root'
    lastChanged='2013-03-19T17:58:01'/>
</iq>
```

For more information, see Node Information.

3.3.4 Get Nodes

This command lets the client get information from multiple nodes at once.

Listing 18: Get Nodes

```
<iq type='get'
  from='client@example.org/client'
  to='concentrator@example.org'
  id='10'>
  <getNodes xmlns='urn:xmpp:iot:concentrators' xml:lang='en'>
    <node sourceId='MeteringTopology' nodeId='Node1'/>
    <node sourceId='MeteringTopology' nodeId='Node2'/>
    <node sourceId='MeteringTopology' nodeId='Node3'/>
  </getNodes>
</iq>

<iq type='result'
  from='concentrator@example.org'
  to='client@example.org/client'
  id='10'>
  <getNodesResponse xmlns='urn:xmpp:iot:concentrators' result='OK'>
    <node nodeId='Node1' nodeType='Namespace.NodeType1' cacheType='
        Node' state='WarningUnsigned' hasChildren='false' isReadable='
        true'
      isControllable='true' hasCommands='true' parentId='Root'
          lastChanged='2013-03-19T17:58:01'/>
    <node nodeId='Node2' nodeType='Namespace.NodeType2' cacheType='
        Node' state='None' hasChildren='false' isReadable='true'
      isControllable='true' hasCommands='true' parentId='Root'
          lastChanged='2013-03-19T17:58:01'/>
    <node nodeId='Node3' nodeType='Namespace.NodeType3' cacheType='
        Node' state='None' hasChildren='false' isReadable='true'
      isControllable='true' hasCommands='true' parentId='Root'
          lastChanged='2013-03-19T17:58:01'/>
  </getNodesResponse>
</iq>
```

For more information, see Node Information.

3.3.5 Get Node with parameters

This command returns basic information about a node in the concentrator, as well as node parameters.

Listing 19: Get Node with parameters

```
<iq type='get'
  from='client@example.org/client'
  to='concentrator@example.org'
  id='11'>
  <getNode xmlns='urn:xmpp:iot:concentrators' sourceId='
     MeteringTopology' nodeId='Node1' xml:lang='en' parameters='true'
     />
</iq>

<iq type='result'
  from='concentrator@example.org'
  to='client@example.org/client'
  id='11'>
  <getNodeResponse xmlns='urn:xmpp:iot:concentrators' result='OK'
     nodeId='Node1' nodeType='Namespace.NodeType1' cacheType='Node'
    state='WarningUnsigned' hasChildren='false' isReadable='true'
       isControllable='true' hasCommands='true'
    parentId='Root' lastChanged='2013-03-19T17:58:01'>
    <string id='id' name='Node␣ID' value='Node1'/>
    <string id='type' name='Node␣Type' value='Watchamacallit␣
       Temperature␣Sensor␣v1.2'/>
    <string id='sn' name='Serial␣Number' value='123456'/>
    <string id='class' name='Node␣Class' value='Temperature'/>
    <string id='meterLoc' name='Meter␣Location' value='P123502-2'/>
    <int id='addr' name='Address' value='123'/>
    <double id='lat' name='Latitude' value='12.345'/>
    <double id='long' name='Longitude' value='123.45'/>
  </getNodeResponse>
</iq>
```

For more information, see Node Information.

3.3.6 Get Nodes with parameters

This command lets the client get information from multiple nodes at once, including node parameters.

Listing 20: Get Nodes with parameters

```
<iq type='get'
  from='client@example.org/client'
  to='concentrator@example.org'
  id='12'>
  <getNodes xmlns='urn:xmpp:iot:concentrators' parameters='true'
     xml:lang='en'>
    <node sourceId='MeteringTopology' nodeId='Node1'/>
```

```
    <node sourceId='MeteringTopology' nodeId='Node2'/>
    <node sourceId='MeteringTopology' nodeId='Node3'/>
  </getNodes>
</iq>

<iq type='result'
  from='concentrator@example.org'
  to='client@example.org/client'
  id='12'>
  <getNodesResponse xmlns='urn:xmpp:iot:concentrators' result='OK'>
    <node nodeId='Node1' nodeType='Namespace.NodeType1' cacheType='
       Node' state='WarningUnsigned' hasChildren='false' isReadable='
       true'
      isControllable='true' hasCommands='true' parentId='Root'
         lastChanged='2013-03-19T17:58:01'>
      <string id='id' name='Node␣ID' value='Node1'/>
      <string id='type' name='Node␣Type' value='Watchamacallit␣
         Temperature␣Sensor␣v1.2'/>
      <string id='sn' name='Serial␣Number' value='123456'/>
      <string id='class' name='Node␣Class' value='Temperature'/>
      <string id='meterLoc' name='Meter␣Location' value='P123502-2'/>
      <int id='addr' name='Address' value='123'/>
      <double id='lat' name='Latitude' value='12.345'/>
      <double id='long' name='Longitude' value='123.45'/>
    </node>
    <node nodeId='Node2' nodeType='Namespace.NodeType2' cacheType='
       Node' state='None' hasChildren='false' isReadable='true'
      isControllable='true' hasCommands='true' parentId='Root'
         lastChanged='2013-03-19T17:58:01'>
      <string id='id' name='Node␣ID' value='Node2'/>
      <string id='type' name='Node␣Type' value='Watchamacallit␣
         Pressure␣Sensor␣v1.2'/>
      <string id='sn' name='Serial␣Number' value='234567'/>
      <string id='class' name='Node␣Class' value='Pressure'/>
      <string id='meterLoc' name='Meter␣Location' value='P668632-6'/>
      <int id='addr' name='Address' value='124'/>
      <double id='lat' name='Latitude' value='12.345'/>
      <double id='long' name='Longitude' value='123.45'/>
    </node>
    <node nodeId='Node3' nodeType='Namespace.NodeType3' cacheType='
       Node' state='None' hasChildren='false' isReadable='true'
      isControllable='true' hasCommands='true' parentId='Root'
         lastChanged='2013-03-19T17:58:01'>
      <string id='id' name='Node␣ID' value='Node3'/>
      <string id='type' name='Node␣Type' value='Watchamacallit␣
         Electricity␣Meter␣v1.2'/>
      <string id='sn' name='Serial␣Number' value='345678'/>
      <string id='class' name='Node␣Class' value='Electricity'/>
      <string id='meterLoc' name='Meter␣Location' value='P332367-9'/>
```

```
      <int id='addr' name='Address' value='125'/>
      <double id='lat' name='Latitude' value='12.345'/>
      <double id='long' name='Longitude' value='123.45'/>
    </node>
  </getNodesResponse>
</iq>
```

For more information, see Node Information.

3.3.7 Get All Nodes

If the device does not manage too many nodes, it could choose to implement this function. It would return all available nodes with one call.

Listing 21: Get All Nodes

```
<iq type='get'
  from='client@example.org/client'
  to='concentrator@example.org'
  id='13'>
  <getAllNodes xmlns='urn:xmpp:iot:concentrators' sourceId='
     MeteringTopology' xml:lang='en'/>
</iq>

<iq type='result'
  from='concentrator@example.org'
  to='client@example.org/client'
  id='13'>
  <getAllNodesResponse xmlns='urn:xmpp:iot:concentrators' result='OK'>
    <node nodeId='Node1' nodeType='Namespace.NodeType1' cacheType='
       Node' state='WarningUnsigned' hasChildren='false'
      isReadable='true' isControllable='true' hasCommands='true'
         parentId='Root'/>
    <node nodeId='Node2' nodeType='Namespace.NodeType2' cacheType='
       Node' state='None' hasChildren='false'
      isReadable='true' isControllable='true' hasCommands='true'
         parentId='Root'/>
    <node nodeId='Node3' nodeType='Namespace.NodeType3' cacheType='
       Node' state='None' hasChildren='false'
      isReadable='true' isControllable='true' hasCommands='true'
         parentId='Root'/>
    <node nodeId='Root' nodeType='Namespace.Root' cacheType='Node'
       state='None' hasChildren='true'
      isReadable='false' isControllable='false' hasCommands='true'/>
  </getAllNodesResponse>
</iq>
```

3.3.8 Get All Nodes with Parameters

If the device does not manage too many nodes, it could choose to implement this function. It would return all available nodes with their parameters with one call.

Listing 22: Get All Nodes with Parameters

```
<iq type='get'
  from='client@example.org/client'
  to='concentrator@example.org'
  id='14'>
  <getAllNodes xmlns='urn:xmpp:iot:concentrators' sourceId='
     MeteringTopology' parameters='true' xml:lang='en'/>
</iq>

<iq type='result'
  from='concentrator@example.org'
  to='client@example.org/client'
  id='14'>
  <getAllNodesResponse xmlns='urn:xmpp:iot:concentrators' result='OK'>
    <node nodeId='Node1' nodeType='Namespace.NodeType1' cacheType='
       Node' state='WarningUnsigned' hasChildren='false'
      isReadable='true' isControllable='true' hasCommands='true'
         parentId='Root'>
      <string id='id' name='Node ID' value='Node1'/>
      <string id='type' name='Node Type' value='Watchamacallit
         Temperature Sensor v1.2'/>
      <string id='sn' name='Serial Number' value='123456'/>
      <string id='class' name='Node Class' value='Temperature'/>
      <string id='meterLoc' name='Meter Location' value='P123502-2'/>
      <int id='addr' name='Address' value='123'/>
      <double id='lat' name='Latitude' value='12.345'/>
      <double id='long' name='Longitude' value='123.45'/>
    </node>
    <node nodeId='Node2' nodeType='Namespace.NodeType2' cacheType='
       Node' state='None' hasChildren='false'
      isReadable='true' isControllable='true' hasCommands='true'
         parentId='Root'>
      <string id='id' name='Node ID' value='Node2'/>
      <string id='type' name='Node Type' value='Watchamacallit
         Pressure Sensor v1.2'/>
      <string id='sn' name='Serial Number' value='234567'/>
      <string id='class' name='Node Class' value='Pressure'/>
      <string id='meterLoc' name='Meter Location' value='P668632-6'/>
      <int id='addr' name='Address' value='124'/>
      <double id='lat' name='Latitude' value='12.345'/>
      <double id='long' name='Longitude' value='123.45'/>
    </node>
    <node nodeId='Node3' nodeType='Namespace.NodeType3' cacheType='
```

```
        Node' state='None' hasChildren='false'
      isReadable='true' isControllable='true' hasCommands='true'
         parentId='Root'>
      <string id='id' name='Node ID' value='Node3'/>
      <string id='type' name='Node Type' value='Watchamacallit
         Electricity Meter v1.2'/>
      <string id='sn' name='Serial Number' value='345678'/>
      <string id='class' name='Node Class' value='Electricity'/>
      <string id='meterLoc' name='Meter Location' value='P332367-9'/>
      <int id='addr' name='Address' value='125'/>
      <double id='lat' name='Latitude' value='12.345'/>
      <double id='long' name='Longitude' value='123.45'/>
    </node>
    <node nodeId='Root' nodeType='Namespace.Root' cacheType='Node'
       state='None' hasChildren='true'
      isReadable='false' isControllable='false' hasCommands='true'>
      <string id='id' name='Node ID' value='Root'/>
      <string id='type' name='Node Type' value='Root Node'/>
    </node>
  </getAllNodesResponse>
</iq>
```

3.3.9 Get All Nodes derived from

This command assumes node types exist in a class hierarchy, and allows the caller to retrieve nodes with similar inheritance.

Listing 23: Get All Nodes derived from

```
<iq type='get'
  from='client@example.org/client'
  to='concentrator@example.org'
  id='15'>
  <getAllNodes xmlns='urn:xmpp:iot:concentrators' sourceId='
     MeteringTopology' xml:lang='en'>
    <onlyIfDerivedFrom>Namespace.BaseClass1</onlyIfDerivedFrom>
  </getAllNodes>
</iq>

<iq type='result'
  from='concentrator@example.org'
  to='client@example.org/client'
  id='15'>
  <getAllNodesResponse xmlns='urn:xmpp:iot:concentrators' result='OK'>
    <node nodeId='Node1' nodeType='Namespace.NodeType1' cacheType='
       Node' state='WarningUnsigned' hasChildren='false'
      isReadable='true' isControllable='true' hasCommands='true'
         parentId='Root'/>
```

```
    <node nodeId='Node3' nodeType='Namespace.NodeType3' cacheType='
       Node' state='None' hasChildren='false'
      isReadable='true' isControllable='true' hasCommands='true'
         parentId='Root'/>
  </getAllNodesResponse>
</iq>
```

3.3.10 Get All Nodes derived from, with Parameters

This command assumes node types exist in a class hierarchy, and allows the caller to retrieve nodes with similar inheritance. It also returns node parameters directly in the response.

Listing 24: Get All Nodes derived from, with Parameters

```
<iq type='get'
  from='client@example.org/client'
  to='concentrator@example.org'
  id='16'>
  <getAllNodes xmlns='urn:xmpp:iot:concentrators' sourceId='
     MeteringTopology' parameters='true' xml:lang='en'>
    <onlyIfDerivedFrom>Namespace.BaseClass1</onlyIfDerivedFrom>
  </getAllNodes>
</iq>

<iq type='result'
  from='concentrator@example.org'
  to='client@example.org/client'
  id='16'>
  <getAllNodesResponse xmlns='urn:xmpp:iot:concentrators' result='OK'>
    <node nodeId='Node1' nodeType='Namespace.NodeType1' cacheType='
       Node' state='WarningUnsigned' hasChildren='false'
      isReadable='true' isControllable='true' hasCommands='true'
         parentId='Root'>
      <string id='id' name='Node␣ID' value='Node1'/>
      <string id='type' name='Node␣Type' value='Watchamacallit␣
         Temperature␣Sensor␣v1.2'/>
      <string id='sn' name='Serial␣Number' value='123456'/>
      <string id='class' name='Node␣Class' value='Temperature'/>
      <string id='meterLoc' name='Meter␣Location' value='P123502-2'/>
      <int id='addr' name='Address' value='123'/>
      <double id='lat' name='Latitude' value='12.345'/>
      <double id='long' name='Longitude' value='123.45'/>
    </node>
    <node nodeId='Node3' nodeType='Namespace.NodeType3' cacheType='
       Node' state='None' hasChildren='false'
      isReadable='true' isControllable='true' hasCommands='true'
         parentId='Root'>
      <string id='id' name='Node␣ID' value='Node3'/>
```

```
      <string id='type' name='Node Type' value='Watchamacallit
         Electricity Meter v1.2'/>
      <string id='sn' name='Serial Number' value='345678'/>
      <string id='class' name='Node Class' value='Electricity'/>
      <string id='meterLoc' name='Meter Location' value='P332367-9'/>
      <int id='addr' name='Address' value='125'/>
      <double id='lat' name='Latitude' value='12.345'/>
      <double id='long' name='Longitude' value='123.45'/>
    </node>
  </getAllNodesResponse>
</iq>
```

Note that the caller can list multiple classes in the request. This would return only nodes having the correct base class(es) and implementing all interfaces.

3.3.11 Get Node Inheritance

This command assumes node types exist in a class hierarchy. It allows the caller to get a list of the node class hierarchy and implemented interfaces the node has.

Listing 25: Get node inheritance

```
<iq type='get'
  from='client@example.org/client'
  to='concentrator@example.org'
  id='17'>
  <getNodeInheritance xmlns='urn:xmpp:iot:concentrators' sourceId='
     MeteringTopology' nodeId='Node1'/>
</iq>

<iq type='result'
  from='concentrator@example.org'
  to='client@example.org/client'
  id='17'>
  <getNodeInheritanceResponse xmlns='urn:xmpp:iot:concentrators'
     result='OK'>
    <baseClasses>
      <value>Namespace.BaseClass1</value>
      <value>Namespace.AbstractBase</value>
    </baseClasses>
  </getNodeInheritanceResponse>
</iq>
```

Note: It is assumed the client already knows the node type of the node, so the response must not contain the type of the node, only its base classes and any implemented interfaces.

3.3.12 Get Root Nodes

This command returns the root node of a data source (in case the source is a tree-shaped data source) or the nodes of a data source (in case the source is flat).

Listing 26: Get Root Nodes

```
<iq type='get'
  from='client@example.org/client'
  to='concentrator@example.org'
  id='18'>
  <getRootNodes xmlns='urn:xmpp:iot:concentrators' sourceId='
     MeteringTopology' xml:lang='en'/>
</iq>

<iq type='result'
  from='concentrator@example.org'
  to='client@example.org/client'
  id='18'>
  <getRootNodesResponse xmlns='urn:xmpp:iot:concentrators' result='OK'
     >
    <node nodeId='Node1' nodeType='Namespace.NodeType1' cacheType='
       Node' state='WarningUnsigned' hasChildren='false'
      isReadable='true' isControllable='true' hasCommands='true'
         parentId='Root'/>
  </getRootNodesResponse>
</iq>
```

3.3.13 Get Root Nodes with Parameters

This command returns the root node of a data source (in case the source is a tree-shaped data source) or the root nodes of a data source (in case the source is flat), and also returns the parameters for the corresponding nodes.

Listing 27: Get Root Nodes with Parameters

```
<iq type='get'
  from='client@example.org/client'
  to='concentrator@example.org'
  id='19'>
  <getRootNodes xmlns='urn:xmpp:iot:concentrators' sourceId='
     MeteringTopology' parameters='true' xml:lang='en'/>
</iq>

<iq type='result'
  from='concentrator@example.org'
  to='client@example.org/client'
  id='19'>
```

```
  <getRootNodesResponse xmlns='urn:xmpp:iot:concentrators' result='OK'
     >
    <node nodeId='Node1' nodeType='Namespace.NodeType1' cacheType='
       Node' state='WarningUnsigned' hasChildren='false'
      isReadable='true' isControllable='true' hasCommands='true'
         parentId='Root'>
      <string id='id' name='Node␣ID' value='Node1'/>
      <string id='type' name='Node␣Type' value='Watchamacallit␣
         Temperature␣Sensor␣v1.2'/>
      <string id='sn' name='Serial␣Number' value='123456'/>
      <string id='class' name='Node␣Class' value='Temperature'/>
      <string id='meterLoc' name='Meter␣Location' value='P123502-2'/>
      <int id='addr' name='Address' value='123'/>
      <double id='lat' name='Latitude' value='12.345'/>
      <double id='long' name='Longitude' value='123.45'/>
    </node>
  </getRootNodesResponse>
</iq>
```

3.3.14 Get Child Nodes

This command returns the child nodes of a node in a data source.

Listing 28: Get Child Nodes

```
<iq type='get'
  from='client@example.org/client'
  to='concentrator@example.org'
  id='20'>
  <getChildNodes xmlns='urn:xmpp:iot:concentrators' sourceId='
     MeteringTopology' nodeId='Node1' xml:lang='en'/>
</iq>

<iq type='result'
  from='concentrator@example.org'
  to='client@example.org/client'
  id='20'>
  <getChildNodesResponse xmlns='urn:xmpp:iot:concentrators' result='OK
     '>
    <node nodeId='Node2' nodeType='Namespace.NodeType2' cacheType='
       Node' state='None' hasChildren='false'
      isReadable='true' isControllable='true' hasCommands='true'
         parentId='Root'/>
    <node nodeId='Node3' nodeType='Namespace.NodeType3' cacheType='
       Node' state='None' hasChildren='false'
      isReadable='true' isControllable='true' hasCommands='true'
         parentId='Root'/>
  </getChildNodesResponse>
```

```
</iq>
```

3.3.15 Get Child Nodes with Parameters

This command returns the child nodes of a node in a data source, and also returns the parameters for the corresponding nodes.

Listing 29: Get Child Nodes with Parameters

```
<iq type='get'
  from='client@example.org/client'
  to='concentrator@example.org'
  id='21'>
  <getChildNodes xmlns='urn:xmpp:iot:concentrators' sourceId='
      MeteringTopology' nodeId='Node1' parameters='true' xml:lang='en'
       />
</iq>

<iq type='result'
  from='concentrator@example.org'
  to='client@example.org/client'
  id='21'>
  <getChildNodesResponse xmlns='urn:xmpp:iot:concentrators' result='OK
      '>
    <node nodeId='Node2' nodeType='Namespace.NodeType2' cacheType='
        Node' state='None' hasChildren='false'
      isReadable='true' isControllable='true' hasCommands='true'
          parentId='Root'>
      <string id='id' name='Node ID' value='Node2'/>
      <string id='type' name='Node Type' value='Watchamacallit
          Pressure Sensor v1.2'/>
      <string id='sn' name='Serial Number' value='234567'/>
      <string id='class' name='Node Class' value='Pressure'/>
      <string id='meterLoc' name='Meter Location' value='P668632-6'/>
      <int id='addr' name='Address' value='124'/>
      <double id='lat' name='Latitude' value='12.345'/>
      <double id='long' name='Longitude' value='123.45'/>
    </node>
    <node nodeId='Node3' nodeType='Namespace.NodeType3' cacheType='
        Node' state='None' hasChildren='false'
      isReadable='true' isControllable='true' hasCommands='true'
          parentId='Root'>
      <string id='id' name='Node ID' value='Node3'/>
      <string id='type' name='Node Type' value='Watchamacallit
          Electricity Meter v1.2'/>
      <string id='sn' name='Serial Number' value='345678'/>
      <string id='class' name='Node Class' value='Electricity'/>
      <string id='meterLoc' name='Meter Location' value='P332367-9'/>
```

```
      <int id='addr' name='Address' value='125'/>
      <double id='lat' name='Latitude' value='12.345'/>
      <double id='long' name='Longitude' value='123.45'/>
    </node>
  </getChildNodesResponse>
</iq>
```

3.3.16 Get Indices of Data Source

This command returns a list of available indices in a data source. Indices can be used for efficient node look-up.

Listing 30: Get Indices of Data Source

```
<iq type='get'
  from='client@example.org/client'
  to='concentrator@example.org'
  id='22'>
  <getIndices xmlns='urn:xmpp:iot:concentrators' sourceId='
     MeteringGroups' />
</iq>

<iq type='result'
  from='concentrator@example.org'
  to='client@example.org/client'
  id='22'>
  <getIndicesResponse xmlns='urn:xmpp:iot:concentrators' result='OK'>
    <value>country</value>
    <value>region</value>
    <value>city</value>
    <value>area</value>
    <value>street</value>
    <value>building</value>
    <value>apartment</value>
    <value>oid</value>
  </getIndicesResponse>
</iq>
```

3.3.17 Get Nodes from index

This command can be used to get a node or nodes from a data source using an index and an index value.

Listing 31: Get Nodes from index

```
<iq type='get'
  from='client@example.org/client'
```

```
  to='concentrator@example.org'
  id='23'>
  <getNodesFromIndex xmlns='urn:xmpp:iot:concentrators' sourceId='
     MeteringGroups' index='apartment' indexValue='A1-1' xml:lang='en
     '/>
</iq>

<iq type='result'
  from='concentrator@example.org'
  to='client@example.org/client'
  id='23'>
  <getNodesFromIndexResponse xmlns='urn:xmpp:iot:concentrators' result
     ='OK'>
    <node nodeId='Node2' nodeType='Namespace.MeteringTopologyReference
       ' state='None' hasChildren='false'
      isReadable='true' isControllable='true' hasCommands='true'
         parentId='Apartment 1-1'/>
  </getNodesFromIndexResponse>
</iq>
```

3.3.18 Get Nodes from index with Parameters

This command can be used to get a node or nodes from a data source using an index and an index value, and also returns the parameters for the corresponding nodes.

Listing 32: Get Nodes from index with Parameters

```
<iq type='get'
  from='client@example.org/client'
  to='concentrator@example.org'
  id='24'>
  <getNodesFromIndex xmlns='urn:xmpp:iot:concentrators' sourceId='
     MeteringGroups' index='apartment' indexValue='A1-1'
    parameters='true' xml:lang='en'/>
</iq>

<iq type='result'
  from='concentrator@example.org'
  to='client@example.org/client'
  id='24'>
  <getNodesFromIndexResponse xmlns='urn:xmpp:iot:concentrators' result
     ='OK'>
    <node nodeId='Node2' nodeType='Namespace.MeteringTopologyReference
       ' state='None' hasChildren='false'
      isReadable='true' isControllable='true' hasCommands='true'
         parentId='Apartment 1-1'>
      <string id='id' name='Node ID' value='Node2'/>
```

```
      <string id='type' name='Node Type' value='Metering Topology
         Reference'/>
      <string id='referenceId' name='Reference ID' value='Node2'/>
    </node>
  </getNodesFromIndexResponse>
</iq>
```

3.3.19 Get Nodes from indices

This command can be used to get nodes from a set of data source using indices and index values.

Listing 33: Get Nodes from indices

```
<iq type='get'
  from='client@example.org/client'
  to='concentrator@example.org'
  id='25'>
  <getNodesFromIndices xmlns='urn:xmpp:iot:concentrators' xml:lang='en
     '>
    <indexRef sourceId='MeteringGroups' index='apartment' indexValue='
       A1-1'/>
    <indexRef sourceId='MeteringGroups' index='apartment' indexValue='
       A1-2'/>
  </getNodesFromIndices>
</iq>

<iq type='result'
  from='concentrator@example.org'
  to='client@example.org/client'
  id='25'>
  <getNodesFromIndicesResponse xmlns='urn:xmpp:iot:concentrators'
     result='OK'>
    <node nodeId='Node2' nodeType='Namespace.MeteringTopologyReference
       ' state='None' hasChildren='false'
      isReadable='true' isControllable='true' hasCommands='true'
         parentId='Apartment 1-1'/>
    <node nodeId='Node3' nodeType='Namespace.MeteringTopologyReference
       ' state='None' hasChildren='false'
      isReadable='true' isControllable='true' hasCommands='true'
         parentId='Apartment 1-2'/>
  </getNodesFromIndicesResponse>
</iq>
```

3.3.20 Get Nodes from indices with Parameters

This command can be used to get nodes from a set of data source using indices and index values, and also returns the parameters for the corresponding nodes.

Listing 34: Get Nodes from indices with Parameters

```
<iq type='get'
  from='client@example.org/client'
  to='concentrator@example.org'
  id='26'>
  <getNodesFromIndices xmlns='urn:xmpp:iot:concentrators' parameters='
     true' xml:lang='en'>
    <indexRef sourceId='MeteringGroups' index='apartment' indexValue='
       A1-1'/>
    <indexRef sourceId='MeteringGroups' index='apartment' indexValue='
       A1-2'/>
  </getNodesFromIndices>
</iq>

<iq type='result'
  from='concentrator@example.org'
  to='client@example.org/client'
  id='26'>
  <getNodesFromIndicesResponse xmlns='urn:xmpp:iot:concentrators'
     result='OK'>
    <node nodeId='Node2' nodeType='Namespace.MeteringTopologyReference
       ' state='None' hasChildren='false'
      isReadable='true' isControllable='true' hasCommands='true'
         parentId='Apartment␣1-1'>
      <string id='id' name='Node␣ID' value='Node2'/>
      <string id='type' name='Node␣Type' value='Metering␣Topology␣
         Reference'/>
      <string id='referenceId' name='Reference␣ID' value='Node2'/>
    </node>
    <node nodeId='Node3' nodeType='Namespace.MeteringTopologyReference
       ' state='None' hasChildren='false'
      isReadable='true' isControllable='true' hasCommands='true'
         parentId='Apartment␣1-2'>
      <string id='id' name='Node␣ID' value='Node3'/>
      <string id='type' name='Node␣Type' value='Metering␣Topology␣
         Reference'/>
      <string id='referenceId' name='Reference␣ID' value='Node3'/>
    </node>
  </getNodesFromIndicesResponse>
</iq>
```

3.3.21 Get All Index Values

This command can be used to get a list of available index values, given a data source and an index.

Listing 35: Get All Index Values

```
<iq type='get'
  from='client@example.org/client'
  to='concentrator@example.org'
  id='27'>
  <getAllIndexValues xmlns='urn:xmpp:iot:concentrators' sourceId='
     MeteringGroups' index='apartment'/>
</iq>

<iq type='result'
  from='concentrator@example.org'
  to='client@example.org/client'
  id='27'>
  <getAllIndexValuesResponse xmlns='urn:xmpp:iot:concentrators' result
     ='OK'>
    <value>A1-1</value>
    <value>A1-2</value>
    ...
  </getAllIndexValuesResponse>
</iq>
```

3.3.22 Get Node Ancestors

In a tree formed data source, all nodes except the root node has a parent node. The **getAncestors** command allows the client to get a list of all ancestors (parent, grand parent, etc.) of a node, as is shown in the following example:

Listing 36: Get Node Ancestors

```
<iq type='get'
  from='client@example.org/client'
  to='concentrator@example.org'
  id='42'>
  <getAncestors xmlns='urn:xmpp:iot:concentrators' sourceId='
     MeteringGroups' nodeId='Node2' parameters='false' messages='
     false' xml:lang='en'/>
</iq>

<iq type='result'
  from='concentrator@example.org'
  to='client@example.org/client'
  id='42'>
  <getAncestorsResponse xmlns='urn:xmpp:iot:concentrators' result='OK'
     >
    <node nodeId='Node2' nodeType='Namespace.MeteringTopologyReference
       ' state='None' hasChildren='false'
      isReadable='true' isControllable='true' hasCommands='true'
         parentId='Apartment_1-1'/>
```

```
    <node nodeId='Apartment 1-1' nodeType='Namespace.Apartment' state=
        'None' hasChildren='true'
      isReadable='false' isControllable='false' hasCommands='true'
         parentId='Building 1'/>
    <node nodeId='Building 1' nodeType='Namespace.Building' state='
        None' hasChildren='true'
      isReadable='false' isControllable='false' hasCommands='true'
         parentId='Street'/>
    <node nodeId='Street' nodeType='Namespace.Street' state='None'
        hasChildren='true'
      isReadable='false' isControllable='false' hasCommands='true'
         parentId='Area'/>
    <node nodeId='Area' nodeType='Namespace.Area' state='None'
        hasChildren='true'
      isReadable='false' isControllable='false' hasCommands='true'
         parentId='City'/>
    <node nodeId='City' nodeType='Namespace.City' state='None'
        hasChildren='true'
      isReadable='false' isControllable='false' hasCommands='true'
         parentId='Region'/>
    <node nodeId='Region' nodeType='Namespace.Region' state='None'
        hasChildren='true'
      isReadable='false' isControllable='false' hasCommands='true'
         parentId='Country'/>
    <node nodeId='Country' nodeType='Namespace.Country' state='None'
        hasChildren='true'
      isReadable='false' isControllable='false' hasCommands='true'
         parentId='Root'/>
    <node nodeId='Root' nodeType='Namespace.Root' state='None'
        hasChildren='true'
      isReadable='false' isControllable='false' hasCommands='true'/>
  </getAncestorsResponse>
</iq>
```

Note that the concentrator returns information about the node itself in the response. The **parameters** and **messages** attributes are used in the request to control if the concentrator should return node parameters and node status messages in the response as well.

3.3.23 Move Node Up

As the order of siblings in a tree can be important, depending on the context and type of nodes involved, the client may be allowed to move nodes up and down among siblings. To move a node upwards among its siblings is done using the command **moveNodeUp**, as is shown in the following example:

Listing 37: Move Node Up

```
<iq type='set'
  from='client@example.org/client'
  to='concentrator@example.org'
  id='53'>
  <moveNodeUp xmlns='urn:xmpp:iot:concentrators' sourceId='
      MeteringFieldProcessors' nodeId='LogicalOperator'/>
</iq>

<iq type='result'
  from='concentrator@example.org'
  to='client@example.org/client'
  id='53'>
  <moveNodeUpResponse xmlns='urn:xmpp:iot:concentrators' result='OK'/>
</iq>
```

Note that a node that is first among its siblings will maintain its position. The response to the command must still be **OK**.

3.3.24 Move Node Down

As the order of siblings in a tree can be important, depending on the context and type of nodes involved, the client may be allowed to move nodes up and down among siblings. To move a node downwards among its siblings is done using the command **moveNodeDown**, as is shown in the following example:

Listing 38: Move Node Up

```
<iq type='set'
  from='client@example.org/client'
  to='concentrator@example.org'
  id='54'>
  <moveNodeDown xmlns='urn:xmpp:iot:concentrators' sourceId='
      MeteringFieldProcessors' nodeId='LogicalOperator'/>
</iq>

<iq type='result'
  from='concentrator@example.org'
  to='client@example.org/client'
  id='54'>
  <moveNodeDownResponse xmlns='urn:xmpp:iot:concentrators' result='OK'
      />
</iq>
```

Note that a node that is last among its siblings will maintain its position. The response to the command must still be **OK**.

3.3.25 Move Nodes Up

To move a set of nodes upwards among its siblings is done using the command **moveNodesUp**, as is shown in the following example:

Listing 39: Move Nodes Up

```
<iq type='set'
  from='client@example.org/client'
  to='concentrator@example.org'
  id='55'>
  <moveNodesUp xmlns='urn:xmpp:iot:concentrators'>
    <node sourceId='MeteringFieldProcessors' nodeId='LogicalOperator3'
       />
    <node sourceId='MeteringFieldProcessors' nodeId='LogicalOperator4'
       />
    <node sourceId='MeteringFieldProcessors' nodeId='LogicalOperator5'
       />
  </moveNodesUp>
</iq>

<iq type='result'
  from='concentrator@example.org'
  to='client@example.org/client'
  id='55'>
  <moveNodesUpResponse xmlns='urn:xmpp:iot:concentrators' result='OK'/
     >
</iq>
```

Note that a node that is first among its siblings will maintain its position. The response to the command must still be **OK**. If an attempt is performed to move a sequence of nodes that are together first as siblings, none of the nodes move relative to each other.

3.3.26 Move Nodes Down

To move a set of nodes downwards among its siblings is done using the command **moveNodesDown**, as is shown in the following example:

Listing 40: Move Node Down

```
<iq type='set'
  from='client@example.org/client'
  to='concentrator@example.org'
  id='56'>
  <moveNodesDown xmlns='urn:xmpp:iot:concentrators'>
    <node sourceId='MeteringFieldProcessors' nodeId='LogicalOperator3'
       />
```

```
    <node sourceId='MeteringFieldProcessors' nodeId='LogicalOperator4'
        />
    <node sourceId='MeteringFieldProcessors' nodeId='LogicalOperator5'
        />
  </moveNodesDown>
</iq>

<iq type='result'
  from='concentrator@example.org'
  to='client@example.org/client'
  id='56'>
  <moveNodesDownResponse xmlns='urn:xmpp:iot:concentrators' result='OK
      '/>
</iq>
```

Note that a node that is last among its siblings will maintain its position. The response to the command must still be **OK**. If an attempt is performed to move a sequence of nodes that are together last as siblings, none of the nodes move relative to each other.

3.4 Node Parameters

3.4.1 Get Node Parameters for editing

Previously described commands can return parameters for a node. But these parameters are for presentational or informational use. If the client wants to edit the parameters of a node, another set of commands must be used. This use case shows how **getNodeParametersForEdit** can be used to edit available parameters for one node.
Note: When editing parameters for a node, a different set of parameters might be returned compared to the set of parameters available in commands mentioned above. There may be various reasons for this, among other things (but not limited to) user rights, node settings, and parameter type. User rights may restrict the number of parameters the user can access. The node may be configured not to allow editing of certain parameters. Also, some types of parameters may only be available in an edit mode (like long multi-line parameters) and not in a shorter presentation mode.

Listing 41: Get Node Parameters for editing

```
<iq type='get'
            from='client@example.org/client'
            to='concentrator@example.org'
            id='28'>
            <getNodeParametersForEdit xmlns='
                urn:xmpp:iot:concentrators' sourceId='MeteringTopology
                ' nodeId='Node1' xml:lang='en'/>
          </iq>

          <iq type='result'
```

```
from='concentrator@example.org'
to='client@example.org/client'
id='28'>
<getNodeParametersForEditResponse xmlns='
    urn:xmpp:iot:concentrators' result='OK'>
  <x type='form'
    xmlns='jabber:x:data'
    xmlns:xdv='http://jabber.org/protocol/xdata-validate'
    xmlns:xdl='http://jabber.org/protocol/xdata-layout'
    xmlns:xdd='urn:xmpp:xdata:dynamic'>
    <title>Node1</title>
    <xdl:page label='Identity'>
      <xdl:fieldref var='id'/>
      <xdl:fieldref var='type'/>
      <xdl:fieldref var='class'/>
      <xdl:fieldref var='sn'/>
    </xdl:page>
    <xdl:page label='Location'>
      <xdl:fieldref var='meterLoc'/>
      <xdl:fieldref var='lat'/>
      <xdl:fieldref var='long'/>
    </xdl:page>
    <xdl:page label='Communication'>
      <xdl:fieldref var='addr'/>
    </xdl:page>
    <field var='xdd␣session' type='hidden'>
      <value>009c7956-001c-43fb-8edb-76bcf74272c9</value>
    </field>
    <field var='id' type='text-single' label='Node␣ID:'>
      <desc>ID of the node.</desc>
      <required/>
      <value>Node1</value>
    </field>
    <field var='type' type='text-single' label='Node␣Type:
        '>
      <desc>Type of node.</desc>
      <value>Watchamacallit Temperature Sensor v1.2</value
          >
      <xdd:readOnly/>
    </field>
    <field var='class' type='list-single' label='Node␣
        Class:'>
      <desc>Class of node</desc>
      <value>Temperature</value>
      <option label='Cooling'><value>Cooling</value></
          option>
      <option label='Electricity'><value>Electricity</
          value></option>
      <option label='Heating'><value>Heating</value></
```

```
                option>
              <option label='Pressure'><value>Pressure</value></
                option>
              <option label='Temperature'><value>Temperature</
                value></option>
              <option label='Water'><value>Water</value></option>
              ...
            </field>
            <field var='sn' type='text-single' label='Serial␣
              Number:'>
              <desc>Serial number of node/device.</desc>
              <value>123456</value>
            </field>
            <field var='meterLoc' type='text-single' label='Meter␣
              Location:'>
              <desc>Meter Location.</desc>
              <value>P123502-2</value>
            </field>
            <field var='addr' type='text-single' label='Address:'>
              <xdv:validate datatype='xs:int'>
                <xdv:range min='1' max='250'/>
              </xdv:validate>
              <desc>Bus address</desc>
              <value>123</value>
            </field>
            <field var='lat' type='text-single' label='Latitude:'>
              <xdv:validate datatype='xs:double'>
                <xdv:range min='-90' max='90'/>
              </xdv:validate>
              <desc>Latitude of node.</desc>
              <value>12.345</value>
            </field>
            <field var='long' type='text-single' label='Longitude:
              '>
              <xdv:validate datatype='xs:double'>
                <xdv:range min='-180' max='180'/>
              </xdv:validate>
              <desc>Longitude of node.</desc>
              <value>123.45</value>
            </field>
          </x>
        </getNodeParametersForEditResponse>
      </iq>
```

The following table lists the different XEP's the client should implement to be able to support parameter forms according to this proposal:

XEP	Description
XEP-0004	Describes how basic forms are handled.
XEP-0122	Makes it possible to add certain client validation rules to form parameters.
XEP-0137	Makes it possible to publish a file upload parameter.
XEP-0141	Makes it possible to layout parameters into pages and sections.
XEP-0331	Defines extensions for how color parameters can be handled.
XEP-0336	Makes it possible to create dynamic forms, with server-side validation and forms that change dynamically depending on input.

Read-only parameters will be returned with the **readOnly** element, as defined in XEP-0336. Clients SHOULD support this extension if using this command. However, the server MUST NOT change parameters in a node that are read-only, even if clients happen to try to set them.

3.4.2 Set Node Parameters after editing

After editing the form, the client uses the **setNodeParametersAfterEdit** command to set the parameters in the node. Note that it is possible to set the same parameters (or a sub-set of the same parameters) to a different node using this command, without the need to get new form parameters. However, after the first successful set operation, any form session used for dynamic validation during edit will not be available on the server anymore and must be ignored by the server.

Listing 42: Set Node Parameters after editing

```
<iq type='set'
            from='client@example.org/client'
            to='concentrator@example.org'
            id='29'>
            <setNodeParametersAfterEdit xmlns='
                urn:xmpp:iot:concentrators' sourceId='MeteringTopology
                ' nodeId='Node1' xml:lang='en'>
              <x type='submit' xmlns='jabber:x:data'>
                <field var='xdd_session' type='hidden'>
                  <value>009c7956-001c-43fb-8edb-76bcf74272c9</value>
                </field>
                <field var='id' type='text-single'>
                  <value>Node1</value>
                </field>
                <field var='class' type='list-single'>
                  <value>Temperature</value>
                </field>
                <field var='sn' type='text-single'>
                  <value>123456</value>
                </field>
```

```
            <field var='meterLoc' type='text-single'>
              <value>P123502-2</value>
            </field>
            <field var='addr' type='text-single'>
              <value>123</value>
            </field>
            <field var='lat' type='text-single'>
              <value>12.345</value>
            </field>
            <field var='long' type='text-single'>
              <value>123.45</value>
            </field>
          </x>
        </setNodeParametersAfterEdit>
      </iq>

      <iq type='result'
        from='concentrator@example.org'
        to='client@example.org/client'
        id='29'>
        <setNodeParametersAfterEditResponse xmlns='
            urn:xmpp:iot:concentrators' result='OK'>
          <node nodeId='Node1' nodeType='Namespace.NodeType1'
              cacheType='Node' state='WarningUnsigned' hasChildren
              ='false'
            isReadable='true' isControllable='true' hasCommands='
                true' parentId='Root'>
            <string id='id' name='Node ID' value='Node1'/>
            <string id='type' name='Node Type' value='
                Watchamacallit Temperature Sensor v1.2'/>
            <string id='sn' name='Serial Number' value='123456'/>
            <string id='class' name='Node Class' value='
                Temperature'/>
            <string id='meterLoc' name='Meter Location' value='
                P123502-2'/>
            <int id='addr' name='Address' value='123'/>
            <double id='lat' name='Latitude' value='12.345'/>
            <double id='long' name='Longitude' value='123.45'/>
          </node>
        </setNodeParametersAfterEditResponse>
      </iq>
```

Note that validation rules, pagination, etc., can be stripped from the form when submitting it to the server. Also the form type attribute must be set to **'submit'**. Note also that as the **result** attribute is **OK**, it is assumed the server has dropped any parameter form resources related to the form, which disables any future dynamic validation of the contents of the form. The newly edited node will also be available in the response in a **node** element.

3.4.3 Set Node Parameters after editing, Failure

The following example shows how the server responds when the client tries to set invalid parameters. The response contains detailed information about why, information which the client can use to inform the user (if any) of what went wrong.

Listing 43: Set Node Parameters after editing, Failure

```
<iq type='set'
            from='client@example.org/client'
            to='concentrator@example.org'
            id='30'>
            <setNodeParametersAfterEdit xmlns='
                urn:xmpp:iot:concentrators' sourceId='MeteringTopology
                ' nodeId='Node2' xml:lang='en'>
              <x type='submit' xmlns='jabber:x:data'>
                <field var='xdd_session' type='hidden'>
                  <value>009c7956-001c-43fb-8edb-76bcf74272c9</value>
                </field>
                <field var='id' type='text-single'>
                  <value>Node1</value>
                </field>
                <field var='class' type='list-single'>
                  <value>Temperature</value>
                </field>
                <field var='sn' type='text-single'>
                  <value>123456</value>
                </field>
                <field var='meterLoc' type='text-single'>
                  <value>P123502-2</value>
                </field>
                <field var='addr' type='text-single'>
                  <value>123</value>
                </field>
                <field var='lat' type='text-single'>
                  <value>12.345</value>
                </field>
                <field var='long' type='text-single'>
                  <value>123.45</value>
                </field>
              </x>
            </setNodeParametersAfterEdit>
          </iq>

          <iq type='error'
            from='concentrator@example.org'
            to='client@example.org/client'
            id='30'>
            <setNodeParametersAfterEditResponse xmlns='
```

```
            urn:xmpp:iot:concentrators' result='FormError'>
          <error var='id'>There already exists a node with this ID
            .</error>
        </setNodeParametersAfterEditResponse>
      </iq>
```

As the **result** attribute is **FormError**, the server maintains any parameter form resources related to the form, and features such as dynamic validation of the contents of the form will still be available until the parameters have been successfully set, the operation has been explicitly cancelled or a form session time-out has occurred. See XEP-0336 [4] for more information.

3.4.4 Get Common Node Parameters for editing

Advanced concentrators handling large quantities of nodes may let users edit sets of nodes at once to be practical. This is done by publishing the **getCommonNodeParametersForEdit** command. It will return a form with parameters that are common for all selected nodes. Since nodes may have different node types it is assumed that different nodes have different sets of parameters. But if this command is used, only parameters matching in IDs, descriptions, validation rules, etc., (but not values) will be returned in a form.
Important: A parameter that exists in multiple nodes, but has different parameter values among the nodes, will be marked with the **notSame** element, according to XEP-0336. Clients using this command MUST support the extensions defined in XEP-0336.

Listing 44: Get Common Node Parameters for editing

```
<iq type='get'
          from='client@example.org/client'
          to='concentrator@example.org'
          id='31'>
          <getCommonNodeParametersForEdit xmlns='
              urn:xmpp:iot:concentrators' xml:lang='en'>
            <node sourceId='MeteringTopology' nodeId='Node1'/>
            <node sourceId='MeteringTopology' nodeId='Node2'/>
            <node sourceId='MeteringTopology' nodeId='Node3'/>
          </getCommonNodeParametersForEdit>
        </iq>

        <iq type='result'
          from='concentrator@example.org'
          to='client@example.org/client'
          id='31'>
          <getCommonNodeParametersForEditResponse xmlns='
              urn:xmpp:iot:concentrators' result='OK'>
            <x type='form'
```

[4] XEP-0336: Dynamic Data Forms <http://xmpp.org/extensions/xep-0336.html>

```
xmlns='jabber:x:data'
xmlns:xdv='http://jabber.org/protocol/xdata-validate'
xmlns:xdl='http://jabber.org/protocol/xdata-layout'
xmlns:xdd='urn:xmpp:xdata:dynamic'>
<title>Node1, Node2, Node3</title>
<xdl:page label='Identity'>
  <xdl:fieldref var='type'/>
  <xdl:fieldref var='class'/>
  <xdl:fieldref var='sn'/>
</xdl:page>
<xdl:page label='Location'>
  <xdl:fieldref var='meterLoc'/>
  <xdl:fieldref var='lat'/>
  <xdl:fieldref var='long'/>
</xdl:page>
<field var='xdd_session' type='hidden'>
  <value>009c7956-001c-43fb-8edb-76bcf74272c9</value>
</field>
<field var='type' type='text-single' label='Node_Type:
    '>
  <desc>Type of node.</desc>
  <value>Watchamacallit Temperature Sensor v1.2</value
     >
  <xdd:readOnly/>
  <xdd:notSame/>
</field>
<field var='class' type='list-single' label='Node_
    Class:'>
  <desc>Class of node</desc>
  <value>Temperature</value>
  <xdd:notSame/>
  <option label='Cooling'><value>Cooling</value></
     option>
  <option label='Electricity'><value>Electricity</
     value></option>
  <option label='Heating'><value>Heating</value></
     option>
  <option label='Pressure'><value>Pressure</value></
     option>
  <option label='Temperature'><value>Temperature</
     value></option>
  <option label='Water'><value>Water</value></option>
  ...
</field>
<field var='sn' type='text-single' label='Serial_
    Number:'>
  <desc>Serial number of node/device.</desc>
  <value>123456</value>
  <xdd:notSame/>
```

```
            </field>
            <field var='meterLoc' type='text-single' label='Meter␣
                Location:'>
              <desc>Meter Location.</desc>
              <value>P123502-2</value>
              <xdd:notSame/>
            </field>
            <field var='addr' type='text-single' label='Address:'>
              <xdv:validate datatype='xs:int'>
                <xdv:range min='1' max='250'/>
              </xdv:validate>
              <desc>Bus address</desc>
              <value>123</value>
              <xdd:notSame/>
            </field>
            <field var='lat' type='text-single' label='Latitude:'>
              <xdv:validate datatype='xs:double'>
                <xdv:range min='-90' max='90'/>
              </xdv:validate>
              <desc>Latitude of node.</desc>
              <value>12.345</value>
              <xdd:notSame/>
            </field>
            <field var='long' type='text-single' label='Longitude:
                '>
              <xdv:validate datatype='xs:double'>
                <xdv:range min='-180' max='180'/>
              </xdv:validate>
              <desc>Longitude of node.</desc>
              <value>123.45</value>
              <xdd:notSame/>
            </field>
          </x>
        </getCommonNodeParametersForEditResponse>
      </iq>
```

Note that parameters that are not available in all selected nodes will have been removed. Also and ID-parameter will have been removed, since they cannot be set for a collection of nodes. Fields marked with the **notSame** element only present one value, perhaps the value of the first node. However, the field should be clearly marked in any end-user GUI (for example by graying the field), and MUST ONLY be sent back to the server in a set operation if explicitly edited by the end-user. The parameter will be set in all selected nodes in that case. Unedited fields should be treated as if the end-user accepts the different values for the current set of nodes.

3.4.5 Set Common Node Parameters after editing

After editing the form, the client uses the **setCommonNodeParametersAfterEdit** command to set the parameters in the set of nodes. Note that it is possible to set the same parameters (or a sub-set of the same parameters) to a different set of nodes using this command, without the need to get new form parameters. However, after the first successful set operation, any form session used for dynamic validation during edit will not be available on the server any more.

Listing 45: Set Common Node Parameters after editing

```
<iq type='set'
            from='client@example.org/client'
            to='concentrator@example.org'
            id='32'>
            <setCommonNodeParametersAfterEdit xmlns='
                urn:xmpp:iot:concentrators' xml:lang='en'>
              <node sourceId='MeteringTopology' nodeId='Node1'/>
              <node sourceId='MeteringTopology' nodeId='Node2'/>
              <node sourceId='MeteringTopology' nodeId='Node3'/>
              <x type='submit' xmlns='jabber:x:data'>
                <field var='xdd␣session' type='hidden'>
                  <value>009c7956-001c-43fb-8edb-76bcf74272c9</value>
                </field>
                <field var='class' type='list-single'>
                  <value>Temperature</value>
                </field>
              </x>
            </setCommonNodeParametersAfterEdit>
          </iq>

          <iq type='error'
            from='concentrator@example.org'
            to='client@example.org/client'
            id='32'>
            <setCommonNodeParametersAfterEditResponse xmlns='
                urn:xmpp:iot:concentrators' result='FormError'>
              <node nodeId='Node1' nodeType='Namespace.NodeType1'
                  cacheType='Node' state='WarningUnsigned' hasChildren
                  ='false'
                 isReadable='true' isControllable='true' hasCommands='
                    true' parentId='Root'>
                <string id='id' name='Node␣ID' value='Node1'/>
                <string id='type' name='Node␣Type' value='
                    Watchamacallit␣Temperature␣Sensor␣v1.2'/>
                <string id='sn' name='Serial␣Number' value='123456'/>
                <string id='class' name='Node␣Class' value='
                    Temperature'/>
                <string id='meterLoc' name='Meter␣Location' value='
```

```
                P123502-2'/>
              <int id='addr' name='Address' value='123'/>
              <double id='lat' name='Latitude' value='12.345'/>
              <double id='long' name='Longitude' value='123.45'/>
            </node>
            <node nodeId='Node2' nodeType='Namespace.NodeType2'
                cacheType='Node' state='None' hasChildren='false'
              isReadable='true' isControllable='true' hasCommands='
                true' parentId='Root'>
              <string id='id' name='Node ID' value='Node2'/>
              <string id='type' name='Node Type' value='
                Watchamacallit Pressure Sensor v1.2'/>
              <string id='sn' name='Serial Number' value='234567'/>
              <string id='class' name='Node Class' value='
                Temperature'/>
              <string id='meterLoc' name='Meter Location' value='
                P668632-6'/>
              <int id='addr' name='Address' value='124'/>
              <double id='lat' name='Latitude' value='12.345'/>
              <double id='long' name='Longitude' value='123.45'/>
            </node>
            <node nodeId='Node3' nodeType='Namespace.NodeType3'
                cacheType='Node' state='None' hasChildren='false'
              isReadable='true' isControllable='true' hasCommands='
                true' parentId='Root'>
              <string id='id' name='Node ID' value='Node3'/>
              <string id='type' name='Node Type' value='
                Watchamacallit Electricity Meter v1.2'/>
              <string id='sn' name='Serial Number' value='345678'/>
              <string id='class' name='Node Class' value='
                Temperature'/>
              <string id='meterLoc' name='Meter Location' value='
                P332367-9'/>
              <int id='addr' name='Address' value='125'/>
              <double id='lat' name='Latitude' value='12.345'/>
              <double id='long' name='Longitude' value='123.45'/>
            </node>
          </setCommonNodeParametersAfterEditResponse>
        </iq>
```

Note that validation rules, pagination, etc., can be stripped from the form when submitting it to the server. Also the form type attribute must be set to **'submit'**. Note also that as the **result** attribute is **OK**, it is assumed the server has dropped any parameter form resources related to the form, which disables any future dynamic validation of the contents of the form. **Important:** A parameter that exists in multiple nodes, but has different parameter values among the nodes, will be marked with the **notSame** element, according to XEP-0336. Such parameters MUST NOT be sent back to the server unless they have explicitly been edited or signed by the end-user. The value sent back to the server will be set in all nodes.

3.4.6 Set Common Node Parameters after editing, Failure

The following example shows how the server responds when the client tries to set invalid parameters to a set of nodes. The response contains detailed information about why, information which the client can use to inform the user (if any) of what went wrong.

Listing 46: Set Common Node Parameters after editing, Failure

```
<iq type='set'
            from='client@example.org/client'
            to='concentrator@example.org'
            id='33'>
            <setCommonNodeParametersAfterEdit xmlns='
                urn:xmpp:iot:concentrators' sourceId='MeteringTopology
                ' xml:lang='en'>
              <node sourceId='MeteringTopology' nodeId='Node1'/>
              <node sourceId='MeteringTopology' nodeId='Node2'/>
              <node sourceId='MeteringTopology' nodeId='Node3'/>
              <x type='submit' xmlns='jabber:x:data'>
                <field var='xdd_session' type='hidden'>
                  <value>009c7956-001c-43fb-8edb-76bcf74272c9</value>
                </field>
                <field var='id' type='text-single'>
                  <value>Node1</value>
                </field>
                <field var='class' type='list-single'>
                  <value>Temperature</value>
                </field>
              </x>
            </setCommonNodeParametersAfterEdit>
          </iq>

          <iq type='error'
            from='concentrator@example.org'
            to='client@example.org/client'
            id='33'>
            <setCommonNodeParametersAfterEditResponse xmlns='
                urn:xmpp:iot:concentrators' result='FormError'>
              <error var='id'>Parameter not available.</error>
            </setCommonNodeParametersAfterEditResponse>
          </iq>
```

As the **result** attribute is **FormError**, the server maintains any parameter form resources related to the form, and features such as dynamic validation of the contents of the form will still be available until the parameters have been successfully set, the operation has been explicitly cancelled or a form session time-out has occurred. See XEP-0336 for more information.

3.4.7 Get Node Messages

Each node in the concentrator has a **state**. This state is a dynamic run-time state, and therefore not presented as a more static property. This state can be any of the following values, in order of increasing importance:

State	Description
None	Nothing has been reported on the node.
Information	There are informative events reported on the node.
WarningSigned	There are warnings reported on the node. But these warnings have been viewed by an operator.
WarningUnsigned	There are new or unreviewed warnings reported on the node.
ErrorSigned	There are errors reported on the node. But these errors have been viewed by an operator.
ErrorUnsigned	There are new or unreviewed errors reported on the node.

Other types of ”states” are of course possible, such as phase - installation phase, test phase, production phase, etc. - but such ”states” are seen as static and presented as parameters on the node. The purpose of the dynamic state attribute of a node, is to give a dynamic runtime state that has the possibility to change during runtime, which operators must be aware of.
The following commands have an optional attribute **messages**, with which they can ask the server to return any events logged on the node, giving more details of the current state of the node:

- **getNode**
- **getNodes**
- **getChildNodes**
- **getAllNodes**
- **getRootNodes**
- **getNodesFromIndex**
- **getNodesFromIndices**

Listing 47: Get Node Messages

```
<iq type='get'
            from='client@example.org/client'
            to='concentrator@example.org'
            id='34'>
```

```
            <getNode xmlns='urn:xmpp:iot:concentrators' sourceId='
                MeteringTopology' nodeId='Node1' messages='true'
                xml:lang='en'/>
          </iq>

          <iq type='result'
            from='concentrator@example.org'
            to='client@example.org/client'
            id='34'>
            <getNodeResponse xmlns='urn:xmpp:iot:concentrators' result
                ='OK'
              nodeId='Node1' nodeType='Namespace.NodeType1' cacheType=
                  'Node' state='WarningUnsigned' hasChildren='false'
              isReadable='true' isControllable='true' hasCommands='
                  true' parentId='Root' lastChanged='2013-03-19
                  T17:58:01'>
              <message timestamp='2013-03-21T11:06:15' type='
                  WarningUnsigned'
                eventId='ClockWarning'>Internal clock is offset more
                    than 7 seconds.</message>
            </getNodeResponse>
          </iq>
```

The **messages** attribute can be combined with the **parameters** attribute to provide both node parameters and messages in the response.

Listing 48: Get Node with parameters and messages

```
<iq type='get'
            from='client@example.org/client'
            to='concentrator@example.org'
            id='35'>
            <getNode xmlns='urn:xmpp:iot:concentrators' sourceId='
                MeteringTopology' nodeId='Node1' xml:lang='en'
                parameters='true' messages='true'/>
          </iq>

          <iq type='result'
            from='concentrator@example.org'
            to='client@example.org/client'
            id='35'>
            <getNodeResponse xmlns='urn:xmpp:iot:concentrators' result
                ='OK'
              nodeId='Node1' nodeType='Namespace.NodeType1' cacheType=
                  'Node' state='WarningUnsigned' hasChildren='false'
              isReadable='true' isControllable='true' hasCommands='
                  true' parentId='Root' lastChanged='2013-03-19
                  T17:58:01'>
              <string id='id' name='Node␣ID' value='Node1'/>
```

```
            <string id='type' name='Node Type' value='Watchamacallit
                Temperature Sensor v1.2'/>
            <string id='sn' name='Serial Number' value='123456'/>
            <string id='class' name='Node Class' value='Temperature'
                />
            <string id='meterLoc' name='Meter Location' value='
                P123502-2'/>
            <int id='addr' name='Address' value='123'/>
            <double id='lat' name='Latitude' value='12.345'/>
            <double id='long' name='Longitude' value='123.45'/>
            <message timestamp='2013-03-21T11:06:15' type='
                WarningUnsigned'
              eventId='ClockWarning'>Internal clock is offset more
                  than 7 seconds.</message>
          </getNodeResponse>
        </iq>
```

3.5 Creating and Destroying Nodes

3.5.1 Get Addable Node Types

Since nodes are context sensitive, depending on node type and tree structure, before being able to create a new node, it is important to know what types of nodes that can be added to a given node. This is done using the **getAddableNodeTypes** command, as is shown in the following example:

Listing 49: Get Addable Node Types

```
<iq type='get'
          from='client@example.org/client'
          to='concentrator@example.org'
          id='36'>
          <getAddableNodeTypes xmlns='urn:xmpp:iot:concentrators'
              sourceId='MeteringGroups' nodeId='B1' xml:lang='en'/>
        </iq>

        <iq type='result'
          from='concentrator@example.org'
          to='client@example.org/client'
          id='36'>
          <getAddableNodeTypesResponse xmlns='
              urn:xmpp:iot:concentrators' result='OK'>
            <nodeType type='Namespace.Apartment' name='Apartment'/>
            <nodeType type='Namespace.MeteringTopologyReference'
                name='Metering Topology Reference'/>
            <nodeType type='Namespace.Location' name='Service
                Location'/>
          </getAddableNodeTypesResponse>
```

```
          </iq>
```

3.5.2 Get Parameters for New Node

When you know what type of node you want to create, you need to get a set of parameters you need to fill in for the new node, before you can create it. This is done using the **getParametersForNewNode** command, as is shown in the following example:

Listing 50: Get Parameters for New Node

```
<iq type='get'
            from='client@example.org/client'
            to='concentrator@example.org'
            id='37'>
            <getParametersForNewNode xmlns='urn:xmpp:iot:concentrators
                ' sourceId='MeteringGroups' nodeId='B1'
              type='Namespace.MeteringTopologyReference' xml:lang='en'
                 />
          </iq>

          <iq type='result'
            from='concentrator@example.org'
            to='client@example.org/client'
            id='37'>
            <getParametersForNewNodeResponse xmlns='
               urn:xmpp:iot:concentrators' result='OK'>
              <x type='form'
                xmlns='jabber:x:data'
                xmlns:xdv='http://jabber.org/protocol/xdata-validate'
                xmlns:xdl='http://jabber.org/protocol/xdata-layout'
                xmlns:xdd='urn:xmpp:xdata:dynamic'>
                <title>Metering Topology</title>
                <xdl:page label='Identity'>
                  <xdl:fieldref var='id'/>
                  <xdl:fieldref var='referenceId'/>
                </xdl:page>
                <field var='xdd_session' type='hidden'>
                  <value>0B146517-8EA3-4BEC-A2E9-CF3F209D4A5D</value>
                </field>
                <field var='id' type='text-single' label='Node_ID:'>
                  <desc>ID of the node.</desc>
                  <required/>
                  <value/>
                </field>
                <field var='referenceId' type='text-single' label='
                   Metering_Node_ID:'>
                  <desc>ID of the node in the metering topology.</desc
                     >
```

```
                <required/>
                <value/>
              </field>
            </x>
          </getParametersForNewNodeResponse>
        </iq>
```

3.5.3 Create New Node

After editing the form, the client uses the **createNewNode** command to create the new node using the parameters provided in the form.

Listing 51: Create New Node after editing

```
<iq type='set'
          from='client@example.org/client'
          to='concentrator@example.org'
          id='38'>
          <createNewNode xmlns='urn:xmpp:iot:concentrators' sourceId
              ='MeteringGroups' nodeId='B1'
            type='Namespace.MeteringTopologyReference' xml:lang='en'
                >
            <x type='submit' xmlns='jabber:x:data'>
              <field var='xdd_session' type='hidden'>
                <value>0B146517-8EA3-4BEC-A2E9-CF3F209D4A5D</value>
              </field>
              <field var='id' type='text-single'>
                <value>Reference to Node1</value>
              </field>
              <field var='referenceId' type='text-single'>
                <value>Node1</value>
              </field>
            </x>
          </createNewNode>
        </iq>

        <iq type='result'
          from='concentrator@example.org'
          to='client@example.org/client'
          id='38'>
          <createNewNodeResponse xmlns='urn:xmpp:iot:concentrators'
              result='OK'>
            <node nodeId='Reference_to_Node1' nodeType='Namespace.
                MeteringTopologyReference' state='None' hasChildren=
                'false'
              isReadable='true' isControllable='true' hasCommands='
                  true' parentId='B1'>
```

```
                <string id='id' name='Node ID' value='Reference to
                    Node1'/>
                <string id='type' name='Node Type' value='Metering
                    Topology Reference'/>
                <string id='referenceId' name='Reference ID' value='
                    Node1'/>
              </node>
            </createNewNodeResponse>
          </iq>
```

Note that validation rules, pagination, etc., can be stripped from the form when submitting it to the server. Also the form type attribute must be set to **'submit'**. Note also that as the **result** attribute is **OK**, it is assumed the server has dropped any parameter form resources related to the form, which disables any future dynamic validation of the contents of the form. The newly created node with corresponding parameters is also returned in the response in a **node** element.

3.5.4 Create New Node, Failure

The following example shows how the server responds when it cannot accept parameters provided when trying to create a node. The response will contain detailed information about why, information which the client can use to inform the user (if any) of what went wrong.

Listing 52: Create New Node, Failure

```
<iq type='set'
            from='client@example.org/client'
            to='concentrator@example.org'
            id='39'>
            <createNewNode xmlns='urn:xmpp:iot:concentrators' sourceId
                ='MeteringGroups' nodeId='B1'
              type='Namespace.MeteringTopologyReference' xml:lang='en'
                  >
              <x type='submit' xmlns='jabber:x:data'>
                <field var='xdd session' type='hidden'>
                  <value>0B146517-8EA3-4BEC-A2E9-CF3F209D4A5D</value>
                </field>
                <field var='id' type='text-single'>
                  <value>Node2</value>
                </field>
                <field var='referenceId' type='text-single'>
                  <value>NodeX</value>
                </field>
              </x>
            </createNewNode>
          </iq>
```

```
<iq type='error'
  from='concentrator@example.org'
  to='client@example.org/client'
  id='39'>
  <createNewNodeResponse xmlns='urn:xmpp:iot:concentrators'
      result='FormError'>
    <error var='id'>There already exists a node with this ID
       .</error>
    <error var='referenceId'>Referenced node was not found.<
       /error>
  </createNewNodeResponse>
</iq>
```

As the **result** attribute is **FormError**, the server maintains any parameter form resources related to the form, and features such as dynamic validation of the contents of the form will still be available until the parameters have been successfully set, the operation has been explicitly cancelled or a form session time-out has occurred. See XEP-0336 for more information.

3.5.5 Destroy Node

To destroy (remove) a node from the concentrator, the **destroyNode** command is sent, as is shown in the following example:

Listing 53: Destroy Node

```
<iq type='set'
  from='client@example.org/client'
  to='concentrator@example.org'
  id='40'>
  <destroyNode xmlns='urn:xmpp:iot:concentrators' sourceId='
     MeteringGroups' nodeId='B1' xml:lang='en'/>
</iq>

<iq type='result'
  from='concentrator@example.org'
  to='client@example.org/client'
  id='40'>
  <destroyNodeResponse xmlns='urn:xmpp:iot:concentrators'
     result='OK'/>
</iq>
```

Since the **result** attribute in the response is **OK**, the node has been removed.

3.5.6 Destroy Node, Failure

If the **result** attribute in the response is other than **OK**, the node was not removed from the concentrator. The **result** attribute contains the reason why the operation failed, as is shown in the following example:

Listing 54: Destroy Node, Failure

```
<iq type='set'
            from='client@example.org/client'
            to='concentrator@example.org'
            id='41'>
            <destroyNode xmlns='urn:xmpp:iot:concentrators' sourceId='
                MeteringGroups' nodeId='B1' xml:lang='en'/>
          </iq>

          <iq type='error'
            from='concentrator@example.org'
            to='client@example.org/client'
            id='41'>
            <destroyNodeResponse xmlns='urn:xmpp:iot:concentrators'
                result='InsufficientPrivileges'/>
          </iq>
```

3.6 Node Commands

3.6.1 Get Node Commands

Each node can have a context sensitive set of commands available to it. This is shown using the **hasCommands** attribute in the Node Information record describing the corresponding node. If the client wants to get a list of available commands, the **getNodeCommands** command is sent to the concentrator, as is shown in the following example:

Listing 55: Get Node Commands

```
<iq type='get'
            from='client@example.org/client'
            to='concentrator@example.org'
            id='43'>
            <getNodeCommands xmlns='urn:xmpp:iot:concentrators'
                sourceId='MeteringGroups' nodeId='Apartment_1-1'
                xml:lang='en'/>
          </iq>

          <iq type='result'
            from='concentrator@example.org'
            to='client@example.org/client'
            id='43'>
```

```
            <getNodeCommandsResponse xmlns='urn:xmpp:iot:concentrators
                ' result='OK'>
              <command command='knockDoor' name='Knock␣on␣door' type='
                  Simple'
                confirmationString='Are␣you␣sure␣you␣want␣to␣knock␣on␣
                    the␣door?'
                failureString='Unable␣to␣knock␣on␣the␣door.'
                successString='Door␣knocked.'/>
              <command command='scheduleWakeupCall' name='Schedule␣
                  wakeup␣call' type='Parameterized'
                failureString='Unable␣to␣schedule␣the␣wakeup␣call.'
                successString='Wakeup␣call␣scheduled.'/>
              <command command='searchEvents' name='Search␣events...'
                  type='Query'
                failureString='Unable␣to␣search␣for␣events.'
                successString='Search␣for␣events␣started...'/>
            </getNodeCommandsResponse>
          </iq>
```

There are three types of commands available: **Simple**, **Parameterized** and **Query**. **Simple** commands take no parameters, and are therefore simpler to execute. **Parameterized** commands require the client to get a set of parameters for the corresponding command before it can be executed. **Query** commands also require a set of parameters to be executed, but return a response after (or during) execution in an asynchronous fashion. Queries can also be aborted during execution. A Query with an empty parameter set is considered to be a simple query, not requiring a parameter dialog to be shown.
For more information about command attributes, see Node Commands.

3.6.2 Execute Simple Node Command

Executing a simple command is done by sending the **executeNodeCommand** command to the concentrator, as is shown in the following example:

Listing 56: Execute Simple Node Command

```
<iq type='set'
            from='client@example.org/client'
            to='concentrator@example.org'
            id='44'>
            <executeNodeCommand xmlns='urn:xmpp:iot:concentrators'
                sourceId='MeteringGroups' nodeId='Apartment␣1-1'
                command='knockDoor' xml:lang='en'/>
          </iq>

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

```
            id='44'>
            <executeNodeCommandResponse xmlns='
                urn:xmpp:iot:concentrators' result='OK'/>
          </iq>
```

3.6.3 Get Node Command Parameters

To execute a parameterized command or a query on the node, the client first needs to get (and edit) a set of parameters for the command. Getting a set of parameters for a parameterized command is done as follows:

Listing 57: Get Node Command Parameters

```
<iq type='get'
            from='client@example.org/client'
            to='concentrator@example.org'
            id='45'>
            <getCommandParameters xmlns='urn:xmpp:iot:concentrators'
                sourceId='MeteringGroups' nodeId='Apartment 1-1'
              command='scheduleWakeupCall' xml:lang='en'/>
          </iq>

          <iq type='result'
            from='concentrator@example.org'
            to='client@example.org/client'
            id='45'>
            <getCommandParametersResponse xmlns='
                urn:xmpp:iot:concentrators' result='OK'>
              <x type='form'
                xmlns='jabber:x:data'
                xmlns:xdv='http://jabber.org/protocol/xdata-validate'
                xmlns:xdl='http://jabber.org/protocol/xdata-layout'
                xmlns:xdd='urn:xmpp:xdata:dynamic'>
                <title>Schedule wake-up call</title>
                <field var='xdd_session' type='hidden'>
                  <value>E14E330F-8496-46F0-8F40-178808AB13A7</value>
                </field>
                <field var='time' type='text-single' label='Time:'>
                  <desc>Time of the wake-up call.</desc>
                  <required/>
                  <value></value>
                  <xdv:validate datatype='xs:time'/>
                  <xdv:basic/>
                </xdv:validate>
              </field>
              <field var='mode' type='list-single' label='Wake-up
                  mode:'>
                <desc>Type of wake-up call</desc>
```

```
            <value>Soft</value>
            <option label='Soft'><value>Soft</value></option>
            <option label='Normal'><value>Normal</value></option>
            <option label='Harass'><value>Harass</value></option>
          </field>
        </x>
      </getCommandParametersResponse>
    </iq>
```

3.6.4 Execute Parameterized Node Command

Executing a parameterized command is also done by sending the **executeNodeCommand** command to the concentrator, but including the edited form parameters, as is shown in the following example:

Listing 58: Execute Parameterized Node Command

```
<iq type='set'
    from='client@example.org/client'
    to='concentrator@example.org'
    id='46'>
    <executeNodeCommand xmlns='urn:xmpp:iot:concentrators'
        sourceId='MeteringGroups' nodeId='Apartment 1-1'
      command='scheduleWakeupCall' xml:lang='en'>
      <x type='submit' xmlns='jabber:x:data'>
        <field var='xdd_session' type='hidden'>
          <value>E14E330F-8496-46F0-8F40-178808AB13A7</value>
        </field>
        <field var='time' type='text-single'>
          <value>04:30:00</value>
        </field>
        <field var='mode' type='list-single'>
          <value>Harass</value>
        </field>
      </x>
    </executeNodeCommand>
  </iq>

  <iq type='result'
    from='concentrator@example.org'
    to='client@example.org/client'
    id='46'>
    <executeNodeCommandResponse xmlns='
        urn:xmpp:iot:concentrators' result='OK'/>
  </iq>
```

3.6.5 Execute Node Command, Failure

If an error occurs during the execution of a command or if the server rejects the execution of a command, the server returns a response code different from **OK**. If the response code is **FormError**, the server maintains any parameter form resources related to the form, and features such as dynamic validation of the contents of the form will still be available until the parameters have been successfully set, the operation has been explicitly cancelled or a form session time-out has occurred. See XEP-0336 for more information.

Listing 59: Execute Node Command, Failure

```
<iq type='set'
          from='client@example.org/client'
          to='concentrator@example.org'
          id='47'>
          <executeNodeCommand xmlns='urn:xmpp:iot:concentrators'
              sourceId='MeteringGroups' nodeId='Apartment 1-1'
            command='scheduleWakeupCall' xml:lang='en'>
            <x type='submit' xmlns='jabber:x:data'>
              <field var='xdd_session' type='hidden'>
                <value>E14E330F-8496-46F0-8F40-178808AB13A7</value>
              </field>
              <field var='time' type='text-single'>
                <value>04:30:00</value>
              </field>
              <field var='mode' type='list-single'>
                <value>Harass</value>
              </field>
            </x>
          </executeNodeCommand>
        </iq>

        <iq type='error'
          from='concentrator@example.org'
          to='client@example.org/client'
          id='47'>
          <executeNodeCommandResponse xmlns='
              urn:xmpp:iot:concentrators' result='FormError'>
            <error var='mode'>You are not allowed to harass people at
                04:30:00!</error>
          </executeNodeCommandResponse>
        </iq>
```

3.6.6 Execute Node Query

Executing a Node Query also requires the client to get a set of parameters for the query. This is done in the same way as for parametrized commands, as is shown in the following example:

Listing 60: Get Node Query Parameters

```
<iq type='get'
          from='client@example.org/client'
          to='concentrator@example.org'
          id='74'>
          <getCommandParameters xmlns='urn:xmpp:iot:concentrators'
              sourceId='MeteringGroups' nodeId='Apartment 1-1'
            command='searchEvents' xml:lang='en'/>
        </iq>

        <iq type='result'
          from='concentrator@example.org'
          to='client@example.org/client'
          id='74'>
          <getCommandParametersResponse xmlns='
              urn:xmpp:iot:concentrators' result='OK'>
            <x type='form'
              xmlns='jabber:x:data'
              xmlns:xdv='http://jabber.org/protocol/xdata-validate'
              xmlns:xdl='http://jabber.org/protocol/xdata-layout'
              xmlns:xdd='urn:xmpp:xdata:dynamic'>
              <title>Search for events</title>
              <field var='xdd_session' type='hidden'>
                <value>E14E330F-8496-46F0-8F40-178808AB13A7</value>
              </field>
              <field var='from' type='text-single' label='From:'>
                <desc>From what point in time events will be fetched.<
                    /desc>
                <required/>
                <value>2013-04-16T12:58:00</value>
                <xdv:validate datatype='xs:dateTime'/>
                <xdv:basic/>
              </xdv:validate>
            </field>
            <field var='to' type='text-single' label='To:'>
              <desc>To what point in time events will be fetched.</
                  desc>
              <required/>
              <value>2013-04-23T12:58:00</value>
              <xdv:validate datatype='xs:dateTime'/>
              <xdv:basic/>
            </xdv:validate>
          </field>
          <field var='searchText' type='text-single' label='Search
              Text:'>
            <desc>Only return events including this text.</desc>
            <required/>
            <value></value>
          </field>
```

```
        </x>
      </getCommandParametersResponse>
    </iq>
```

Executing the query is then done by sending the **executeNodeQuery** command to the concentrator, but including the edited form parameters, as is shown in the following example:

Listing 61: Execute Node Query

```
<iq type='set'
      from='client@example.org/client'
      to='concentrator@example.org'
      id='75'>
      <executeNodeQuery xmlns='urn:xmpp:iot:concentrators' sourceId='
          MeteringGroups' nodeId='Apartment␣1-1' command='searchEvents
          ' xml:lang='en'>
        <x type='submit' xmlns='jabber:x:data'>
          <field var='from' type='text-single'>
            <value>2013-04-16T12:58:00</value>
          </field>
          <field var='to' type='text-single'>
            <value>2013-04-23T12:58:00</value>
          </field>
          <field var='searchText' type='text-single'>
            <value>Harass</value>
          </field>
        </x>
      </executeNodeQuery>
    </iq>

    <iq type='result'
      from='concentrator@example.org'
      to='client@example.org/client'
      id='75'>
      <executeNodeQueryResponse xmlns='urn:xmpp:iot:concentrators'
          result='OK' queryId='Query0001'/>
    </iq>

    ...
```

After the successful execution of a query command, a sequence of query events will follow. These events will include a **queryId** attribute to identify to which query the corresponding events correspond. See section about Query Events for more information about this.
Note: Queries with no visible parameters in the command parameter form need not display a parameter form to the user before being executed. However, if a confirmation question is defined for the command, such a confirmation question should always be presented to the user (if possible) before executing the command.

3.6.7 Execute Node Query, Failure

If an error occurs during the reception of a query or if the server rejects the execution of a query, the server returns a response code different from **OK**. If the response code is **FormError**, the server maintains any parameter form resources related to the form, and features such as dynamic validation of the contents of the form will still be available until the parameters have been successfully set, the operation has been explicitly cancelled or a form session time-out has occurred. See XEP-0336 for more information.

Listing 62: Execute Node Query, Failure

```
<iq type='set'
      from='client@example.org/client'
      to='concentrator@example.org'
      id='76'>
      <executeNodeQuery xmlns='urn:xmpp:iot:concentrators' sourceId='
          MeteringGroups' nodeId='Apartment␣1-1' command='searchEvents
          ' xml:lang='en'>
        <x type='submit' xmlns='jabber:x:data'>
          <field var='from' type='text-single'>
            <value>2013-04-23T12:58:00</value>
          </field>
          <field var='to' type='text-single'>
            <value>2013-04-16T12:58:00</value>
          </field>
          <field var='searchText' type='text-single'>
            <value>Harass</value>
          </field>
        </x>
      </executeNodeQuery>
    </iq>

    <iq type='error'
      from='concentrator@example.org'
      to='client@example.org/client'
      id='76'>
      <executeNodeQueryResponse xmlns='urn:xmpp:iot:concentrators'
          result='FormError'>
        <error var='to'>The TO timestamp needs to be later than the
            FROM timestamp.</error>
      </executeNodeQueryResponse>
    </iq>
```

3.6.8 Abort Node Query

One a query has been accepted and started running, the client can abort it using the **abortNodeQuery** command, as is shown in the following example.

Listing 63: Abort Node Query

```
<iq type='set'
      from='client@example.org/client'
      to='concentrator@example.org'
      id='77'>
      <abortNodeQuery xmlns='urn:xmpp:iot:concentrators' sourceId='
          MeteringGroups' nodeId='Apartment 1-1' command='searchEvents
          '
        queryId='Query0001' xml:lang='en'/>
    </iq>

    <iq type='result'
      from='concentrator@example.org'
      to='client@example.org/client'
      id='77'>
      <abortNodeQueryResponse xmlns='urn:xmpp:iot:concentrators'
          result='OK'/>
    </iq>
```

Note: The execution of a query is an asynchronous process, with a small delay between the compilation and transmission of query events and the reception of them by a client. A client may thinkt a query is still active when asking to abort it, when the query might actually have been finished and removed on the query side. Therefore, clients should be aware of this when receiving **NotFound** responses from a concentrator, that the query might already have been finished.

3.6.9 Get Common Commands for Nodes

Using the command **getCommonNodeCommands**, the client can receive commands that are common for a set of nodes, as is shown in the following example:

Listing 64: Get Common Commands for Nodes

```
<iq type='get'
      from='client@example.org/client'
      to='concentrator@example.org'
      id='48'>
      <getCommonNodeCommands xmlns='urn:xmpp:iot:concentrators'
          xml:lang='en'>
        <node sourceId='MeteringGroups' nodeId='Apartment 1-1'/>
        <node sourceId='MeteringGroups' nodeId='Apartment 1-2'/>
        <node sourceId='MeteringGroups' nodeId='Apartment 1-3'/>
        <node sourceId='MeteringGroups' nodeId='Apartment 1-4'/>
        <node sourceId='MeteringGroups' nodeId='Apartment 1-5'/>
        <node sourceId='MeteringGroups' nodeId='Apartment 1-6'/>
      </getCommonNodeCommands>
    </iq>
```

```
<iq type='result'
  from='concentrator@example.org'
  to='client@example.org/client'
  id='48'>
  <getCommonNodeCommandsResponse xmlns='urn:xmpp:iot:concentrators
      ' result='OK'>
    <command command='knockDoor' name='Knock␣on␣door' type='Simple
        '
      confirmationString='Are␣you␣sure␣you␣want␣to␣knock␣on␣the␣
          door?'
      failureString='Unable␣to␣knock␣on␣the␣door.'
      successString='Door␣knocked.'/>
    <command command='scheduleWakeupCall' name='Schedule␣wakeup␣
        call' type='Parameterized'
      failureString='Unable␣to␣schedule␣the␣wakeup␣call.'
      successString='Wakeup␣call␣scheduled.'/>
  </getCommonNodeCommandsResponse>
</iq>
```

3.6.10 Execute Simple Command on multiple nodes

Executing a simple command on multiple nodes is done by sending the **executeCommonNodeCommand** command to the concentrator, as is shown in the following example:

Listing 65: Execute Simple Command on multiple nodes

```
<iq type='set'
    from='client@example.org/client'
    to='concentrator@example.org'
    id='49'>
    <executeCommonNodeCommand xmlns='urn:xmpp:iot:concentrators'
        command='knockDoor' xml:lang='en'>
      <node sourceId='MeteringGroups' nodeId='Apartment␣1-1'/>
      <node sourceId='MeteringGroups' nodeId='Apartment␣1-2'/>
      <node sourceId='MeteringGroups' nodeId='Apartment␣1-3'/>
      <node sourceId='MeteringGroups' nodeId='Apartment␣1-4'/>
      <node sourceId='MeteringGroups' nodeId='Apartment␣1-5'/>
      <node sourceId='MeteringGroups' nodeId='Apartment␣1-6'/>
    </executeCommonNodeCommand>
</iq>

<iq type='result'
  from='concentrator@example.org'
  to='client@example.org/client'
  id='49'>
  <executeCommonNodeCommandResponse xmlns='
      urn:xmpp:iot:concentrators' result='OK'>
```

```
        <result>true</result>
        <result>true</result>
        <result>true</result>
        <result>true</result>
        <result>true</result>
        <result>true</result>
      </executeCommonNodeCommandResponse>
    </iq>
```

3.6.11 Get Common Command Parameters from command on multiple nodes

To execute a parameterized command on a set of nodes, the client first needs to get (and edit) a set of parameters for the common command. Getting a set of parameters for a common parameterized command is done as follows:

Listing 66: Get Common Command Parameters from command on multiple nodes

```
<iq type='get'
      from='client@example.org/client'
      to='concentrator@example.org'
      id='50'>
      <getCommonCommandParameters xmlns='urn:xmpp:iot:concentrators'
          command='scheduleWakeupCall' xml:lang='en'>
        <node sourceId='MeteringGroups' nodeId='Apartment 1-1'/>
        <node sourceId='MeteringGroups' nodeId='Apartment 1-2'/>
        <node sourceId='MeteringGroups' nodeId='Apartment 1-3'/>
        <node sourceId='MeteringGroups' nodeId='Apartment 1-4'/>
        <node sourceId='MeteringGroups' nodeId='Apartment 1-5'/>
        <node sourceId='MeteringGroups' nodeId='Apartment 1-6'/>
      </getCommonCommandParameters>
    </iq>

    <iq type='result'
      from='concentrator@example.org'
      to='client@example.org/client'
      id='50'>
      <getCommonCommandParametersResponse xmlns='
          urn:xmpp:iot:concentrators' result='OK'>
        <x type='form'
          xmlns='jabber:x:data'
          xmlns:xdv='http://jabber.org/protocol/xdata-validate'
          xmlns:xdl='http://jabber.org/protocol/xdata-layout'
          xmlns:xdd='urn:xmpp:xdata:dynamic'>
          <title>Schedule wake-up call</title>
          <field var='xdd_session' type='hidden'>
            <value>E14E330F-8496-46F0-8F40-178808AB13A7</value>
          </field>
          <field var='time' type='text-single' label='Time:'>
```

```
            <desc>Time of the wake-up call.</desc>
            <required/>
            <value></value>
            <xdv:validate datatype='xs:time'/>
            <xdv:basic/>
          </xdv:validate>
        </field>
        <field var='mode' type='list-single' label='Wake-up mode:'>
          <desc>Type of wake-up call</desc>
          <value>Soft</value>
          <option label='Soft'><value>Soft</value></option>
          <option label='Normal'><value>Normal</value></option>
          <option label='Harass'><value>Harass</value></option>
        </field>
      </x>
    </getCommonCommandParametersResponse>
  </iq>
```

3.6.12 Execute Common Parameterized Command on multiple nodes

Executing a common parameterized command is also done by sending the **executeCommonNodeCommand** command to the concentrator, but including the edited form parameters, as is shown in the following example:

Listing 67: Execute Common Parameterized Command on multiple nodes

```
<iq type='set'
                  from='client@example.org/client'
                  to='concentrator@example.org'
                  id='51'>
                  <executeCommonNodeCommand xmlns='
                     urn:xmpp:iot:concentrators' command='
                     scheduleWakeupCall' xml:lang='en'>
                    <node sourceId='MeteringGroups' nodeId='Apartment
                       1-1'/>
                    <node sourceId='MeteringGroups' nodeId='Apartment
                       1-2'/>
                    <node sourceId='MeteringGroups' nodeId='Apartment
                       1-3'/>
                    <node sourceId='MeteringGroups' nodeId='Apartment
                       1-4'/>
                    <node sourceId='MeteringGroups' nodeId='Apartment
                       1-5'/>
                    <node sourceId='MeteringGroups' nodeId='Apartment
                       1-6'/>
                    <x type='submit' xmlns='jabber:x:data'>
                      <field var='xdd session' type='hidden'>
```

```
                    <value>E14E330F-8496-46F0-8F40-178808AB13A7</
                        value>
                  </field>
                  <field var='time' type='text-single'>
                    <value>04:30:00</value>
                  </field>
                  <field var='mode' type='list-single'>
                    <value>Harass</value>
                  </field>
                </x>
              </executeCommonNodeCommand>
            </iq>

            <iq type='result'
              from='concentrator@example.org'
              to='client@example.org/client'
              id='51'>
              <executeCommonNodeCommandResponse xmlns='
                  urn:xmpp:iot:concentrators' result='OK'>
                <result>true</result>
                <result>true</result>
                <result>true</result>
                <result>true</result>
                <result>true</result>
                <result>true</result>
              </executeCommonNodeCommandResponse>
            </iq>
```

3.6.13 Execute Parameterized Command on Multiple Nodes, Failure

If an error occurs during the execution of a common command or if the server rejects the execution of a common command, the server returns a response code different from **OK**. If the response code is **FormError**, the server maintains any parameter form resources related to the form, and features such as dynamic validation of the contents of the form will still be available until the parameters have been successfully set, the operation has been explicitly cancelled or a form session time-out has occurred. See XEP-0336 for more information.

Listing 68: Execute Parameterized Command on Multiple Nodes, Failure

```
<iq type='set'
              from='client@example.org/client'
              to='concentrator@example.org'
              id='52'>
              <executeCommonNodeCommand xmlns='
                  urn:xmpp:iot:concentrators' command='
                  scheduleWakeupCall' xml:lang='en'>
                <node sourceId='MeteringGroups' nodeId='Apartment␣
                    1-1'/>
```

```
        <node sourceId='MeteringGroups' nodeId='Apartment␣
            1-2'/>
        <node sourceId='MeteringGroups' nodeId='Apartment␣
            1-3'/>
        <node sourceId='MeteringGroups' nodeId='Apartment␣
            1-4'/>
        <node sourceId='MeteringGroups' nodeId='Apartment␣
            1-5'/>
        <node sourceId='MeteringGroups' nodeId='Apartment␣
            1-6'/>
        <x type='submit' xmlns='jabber:x:data'>
          <field var='xdd␣session' type='hidden'>
            <value>E14E330F-8496-46F0-8F40-178808AB13A7</
                value>
          </field>
          <field var='time' type='text-single'>
            <value>04:30:00</value>
          </field>
          <field var='mode' type='list-single'>
            <value>Harass</value>
          </field>
        </x>
      </executeCommonNodeCommand>
    </iq>

    <iq type='error'
      from='concentrator@example.org'
      to='client@example.org/client'
      id='52'>
      <executeCommonNodeCommandResponse xmlns='
          urn:xmpp:iot:concentrators' result='FormError'>
        <error var='mode'>You are not allowed to harass
            people at 04:30:00!</error>
      </executeCommonNodeCommandResponse>
    </iq>
```

3.6.14 Execute Command on Multiple Nodes, Partial Failure

When executing a command, simple or parameterized, on multiple nodes, it might happen that the command fails on some nodes, but not on others, even though any parameters are validated beforehand. Therefore, the client needs to check any **result** elements in the response, even though the response code is **OK**.
The following example shows the execution of a parameterized command on multiple nodes that fail on some nodes but are executed on others. Note that individual error messages can be provided as **error** attribute messages for each node that fails in the corresponding **result** element.

Listing 69: Execute Command on Multiple Nodes, Partial Failure

```
<iq type='set'
                    from='client@example.org/client'
                    to='concentrator@example.org'
                    id='72'>
                    <executeCommonNodeCommand xmlns='
                        urn:xmpp:iot:concentrators' command='
                        scheduleWakeupCall' xml:lang='en'>
                      <node sourceId='MeteringGroups' nodeId='Apartment␣
                          1-1'/>
                      <node sourceId='MeteringGroups' nodeId='Apartment␣
                          1-2'/>
                      <node sourceId='MeteringGroups' nodeId='Apartment␣
                          1-3'/>
                      <node sourceId='MeteringGroups' nodeId='Apartment␣
                          1-4'/>
                      <node sourceId='MeteringGroups' nodeId='Apartment␣
                          1-5'/>
                      <node sourceId='MeteringGroups' nodeId='Apartment␣
                          1-6'/>
                      <x type='submit' xmlns='jabber:x:data'>
                        <field var='xdd␣session' type='hidden'>
                          <value>E14E330F-8496-46F0-8F40-178808AB13A7</
                              value>
                        </field>
                        <field var='time' type='text-single'>
                          <value>04:30:00</value>
                        </field>
                        <field var='mode' type='list-single'>
                          <value>Harass</value>
                        </field>
                      </x>
                    </executeCommonNodeCommand>
                  </iq>

                  <iq type='result'
                    from='concentrator@example.org'
                    to='client@example.org/client'
                    id='72'>
                    <executeCommonNodeCommandResponse xmlns='
                        urn:xmpp:iot:concentrators' result='OK'>
                      <result>true</result>
                      <result>true</result>
                      <result error='Too␣early!'>false</result>
                      <result>true</result>
                      <result error='Sleeping␣at␣that␣time...'>false</
                          result>
                      <result>true</result>
                    </executeCommonNodeCommandResponse>
```

```
                </iq>
```

3.6.15 Execute Common Query on multiple nodes

Executing a Node Query on multiple nodes also requires the client to get a set of parameters for the query common to all nodes. This is done in the same way as for parametrized commands, as is shown in the following example:

Listing 70: Get Common Query Parameters from query on multiple nodes

```
<iq type='get'
                  from='client@example.org/client'
                  to='concentrator@example.org'
                  id='78'>
                  <getCommonCommandParameters xmlns='
                     urn:xmpp:iot:concentrators' command='
                     searchEvents' xml:lang='en'>
                    <node sourceId='MeteringGroups' nodeId='Apartment␣
                       1-1'/>
                    <node sourceId='MeteringGroups' nodeId='Apartment␣
                       1-2'/>
                    <node sourceId='MeteringGroups' nodeId='Apartment␣
                       1-3'/>
                    <node sourceId='MeteringGroups' nodeId='Apartment␣
                       1-4'/>
                    <node sourceId='MeteringGroups' nodeId='Apartment␣
                       1-5'/>
                    <node sourceId='MeteringGroups' nodeId='Apartment␣
                       1-6'/>
                  </getCommonCommandParameters>
                </iq>

                <iq type='result'
                  from='concentrator@example.org'
                  to='client@example.org/client'
                  id='78'>
                  <getCommonCommandParametersResponse xmlns='
                     urn:xmpp:iot:concentrators' result='OK'>
                    <x type='form'
                      xmlns='jabber:x:data'
                      xmlns:xdv='http://jabber.org/protocol/xdata-
                         validate'
                      xmlns:xdl='http://jabber.org/protocol/xdata-
                         layout'
                      xmlns:xdd='urn:xmpp:xdata:dynamic'>
                      <title>Schedule wake-up call</title>
                      <field var='xdd␣session' type='hidden'>
```

```
                    <value>E14E330F-8496-46F0-8F40-178808AB13A7</
                        value>
                  </field>
                  <field var='from' type='text-single' label='
                      From:'>
                    <desc>From what point in time events will be
                        fetched.</desc>
                    <required/>
                    <value>2013-04-16T12:58:00</value>
                    <xdv:validate datatype='xs:dateTime'/>
                    <xdv:basic/>
                  </xdv:validate>
                </field>
                <field var='to' type='text-single' label='To:'>
                  <desc>To what point in time events will be
                      fetched.</desc>
                  <required/>
                  <value>2013-04-23T12:58:00</value>
                  <xdv:validate datatype='xs:dateTime'/>
                  <xdv:basic/>
                </xdv:validate>
              </field>
              <field var='searchText' type='text-single' label='
                  Search Text:'>
                <desc>Only return events including this text.</
                    desc>
                <required/>
                <value></value>
              </field>
            </x>
          </getCommonCommandParametersResponse>
```

Executing the query is then done by sending the **executeCommonNodeQuery** command to the concentrator, but including the edited form parameters, as is shown in the following example:

Listing 71: Execute Common Query on multiple nodes

```
<iq type='set'
                from='client@example.org/client'
                to='concentrator@example.org'
                id='79'>
                <executeCommonNodeQuery xmlns='
                    urn:xmpp:iot:concentrators' command='searchEvents'
                     xml:lang='en'>
                  <node sourceId='MeteringGroups' nodeId='Apartment
                      1-1'/>
                  <node sourceId='MeteringGroups' nodeId='Apartment
                      1-2'/>
```

```
            <node sourceId='MeteringGroups' nodeId='Apartment␣
                1-3'/>
            <node sourceId='MeteringGroups' nodeId='Apartment␣
                1-4'/>
            <node sourceId='MeteringGroups' nodeId='Apartment␣
                1-5'/>
            <node sourceId='MeteringGroups' nodeId='Apartment␣
                1-6'/>
            <x type='submit' xmlns='jabber:x:data'>
              <field var='from' type='text-single'>
                <value>2013-04-16T12:58:00</value>
              </field>
              <field var='to' type='text-single'>
                <value>2013-04-23T12:58:00</value>
              </field>
              <field var='searchText' type='text-single'>
                <value>Harass</value>
              </field>
            </x>
          </executeCommonNodeQuery>
        </iq>

        <iq type='result'
          from='concentrator@example.org'
          to='client@example.org/client'
          id='79'>
          <executeCommonNodeQueryResponse xmlns='
              urn:xmpp:iot:concentrators' result='OK' queryId='
              Query0001'/>
        </iq>

        ...
```

After the successful execution of a query command, a sequence of query events will follow. These events will include a **queryId** attribute to identify to which query the corresponding events correspond. They will also include node reference attributes so the receptor can distinguish information from different nodes in the query. See section about Query Events for more information about this.

Note: Queries with no visible parameters in the command parameter form need not display a parameter form to the user before being executed. However, if a confirmation question is defined for the command, such a confirmation question should always be presented to the user (if possible) before executing the command.

3.6.16 Execute Common Query on multiple nodes, Failure

If an error occurs during the execution of a common query or if the server rejects the execution of a common query, the server returns a response code different from **OK**. If

the response code is **FormError**, the server maintains any parameter form resources related to the form, and features such as dynamic validation of the contents of the form will still be available until the parameters have been successfully set, the operation has been explicitly cancelled or a form session time-out has occurred. See XEP-0336 for more information.

Listing 72: Execute Parameterized Query on multiple nodes, Failure

```
<iq type='set'
                from='client@example.org/client'
                to='concentrator@example.org'
                id='80'>
                <executeCommonNodeQuery xmlns='
                    urn:xmpp:iot:concentrators' command='searchEvents'
                     xml:lang='en'>
                  <node sourceId='MeteringGroups' nodeId='Apartment␣
                      1-1'/>
                  <node sourceId='MeteringGroups' nodeId='Apartment␣
                      1-2'/>
                  <node sourceId='MeteringGroups' nodeId='Apartment␣
                      1-3'/>
                  <node sourceId='MeteringGroups' nodeId='Apartment␣
                      1-4'/>
                  <node sourceId='MeteringGroups' nodeId='Apartment␣
                      1-5'/>
                  <node sourceId='MeteringGroups' nodeId='Apartment␣
                      1-6'/>
                  <x type='submit' xmlns='jabber:x:data'>
                    <field var='from' type='text-single'>
                      <value>2013-04-23T12:58:00</value>
                    </field>
                    <field var='to' type='text-single'>
                      <value>2013-04-16T12:58:00</value>
                    </field>
                    <field var='searchText' type='text-single'>
                      <value>Harass</value>
                    </field>
                  </x>
                </executeCommonNodeQuery>
              </iq>

              <iq type='error'
                from='concentrator@example.org'
                to='client@example.org/client'
                id='80'>
                <executeCommonNodeQueryResponse xmlns='
                    urn:xmpp:iot:concentrators' result='FormError'>
                  <error var='to'>The TO timestamp needs to be later
                      than the FROM timestamp.</error>
                </executeCommonNodeQueryResponse>
```

```
</iq>
```

3.6.17 Execute Common Query on multiple nodes, Partial Failure

When executing a query on multiple nodes, it might happen that the query is rejected on some nodes, but not on others, even though any parameters are validated beforehand. Therefore, the client needs to check any **result** elements in the response, even though the response code is **OK**.
The following example shows the execution of a query on multiple nodes that fail on some nodes but are executed on others. Note that individual error messages can be provided as **error** attribute messages for each node that fails in the corresponding **result** element.

Listing 73: Execute Common Query on multiple nodes, Partial Failure

```
<iq type='set'
                from='client@example.org/client'
                to='concentrator@example.org'
                id='81'>
                <executeCommonNodeQuery xmlns='
                    urn:xmpp:iot:concentrators' command='searchEvents'
                     xml:lang='en'>
                  <node sourceId='MeteringGroups' nodeId='Apartment␣
                      1-1'/>
                  <node sourceId='MeteringGroups' nodeId='Apartment␣
                      1-2'/>
                  <node sourceId='MeteringGroups' nodeId='Apartment␣
                      1-3'/>
                  <node sourceId='MeteringGroups' nodeId='Apartment␣
                      1-4'/>
                  <node sourceId='MeteringGroups' nodeId='Apartment␣
                      1-5'/>
                  <node sourceId='MeteringGroups' nodeId='Apartment␣
                      1-6'/>
                  <x type='submit' xmlns='jabber:x:data'>
                    <field var='from' type='text-single'>
                      <value>2013-04-23T12:58:00</value>
                    </field>
                    <field var='to' type='text-single'>
                      <value>2013-04-16T12:58:00</value>
                    </field>
                    <field var='searchText' type='text-single'>
                      <value>Harass</value>
                    </field>
                  </x>
                </executeCommonNodeQuery>
              </iq>
```

```
<iq type='error'
  from='concentrator@example.org'
  to='client@example.org/client'
  id='81'>
  <executeCommonNodeQueryResponse xmlns='
      urn:xmpp:iot:concentrators' result='OK' queryId='
      Query0002'>
    <result>true</result>
    <result>true</result>
    <result error='Event␣Database␣not␣available!'>false<
       /result>
    <result>true</result>
    <result>true</result>
    <result>true</result>
  </executeCommonNodeQueryResponse>
</iq>
```

3.6.18 Abort Common Query on multiple nodes

To abort a query common to a set of nodes you use the **abortCommonNodeQuery** command, as is shown in the following example:

Listing 74: Abort Common Query on multiple nodes

```
<iq type='set'
  from='client@example.org/client'
  to='concentrator@example.org'
  id='82'>
  <abortCommonNodeQuery xmlns='
      urn:xmpp:iot:concentrators' queryId='Query0002'
      xml:lang='en'>
    <node sourceId='MeteringGroups' nodeId='Apartment␣
       1-1'/>
    <node sourceId='MeteringGroups' nodeId='Apartment␣
       1-2'/>
    <node sourceId='MeteringGroups' nodeId='Apartment␣
       1-3'/>
    <node sourceId='MeteringGroups' nodeId='Apartment␣
       1-4'/>
    <node sourceId='MeteringGroups' nodeId='Apartment␣
       1-5'/>
    <node sourceId='MeteringGroups' nodeId='Apartment␣
       1-6'/>
  </abortCommonNodeQuery>
</iq>

<iq type='result'
  from='concentrator@example.org'
```

```
                to='client@example.org/client'
                id='82'>
                <abortCommonNodeQueryResponse xmlns='
                    urn:xmpp:iot:concentrators' result='OK'/>
            </iq>
```

Note: The execution of a query is an asynchronous process, with a small delay between the compilation and transmission of query events and the reception of them by a client. A client may thinkt a query is still active when asking to abort it, when the query might actually have been finished and removed on the query side. Therefore, clients should be aware of this when receiving **NotFound** responses from a concentrator, that the query might already have been finished. In the case of aborting queries on multiple nodes, servers must ignore nodes where the query has already finished executing.

3.7 Query Events

Query events are sent as a response to executing queries on nodes. They are reported asynchronously from the concentrator to the client executing the query. Each event is wrapped in a **queryProgress** element within a message sent from the concentrator to the client. Each such **queryProgress** element can include any number of query events, and the client must process them in the order they appear in the message.
Each **queryProgress** element contains a **queryId** attribute identifying the query to which the events correspond. They also contain **sourceId**, **nodeId** and an optional **cacheType** to identify from which node in a concentrator the events originated.
Note: A **queryProgress** element can only contain events from a single query executed on a single node. If a query is executed on multiple nodes, events resulting from different nodes will be sent in different messages.
The following subsections describe each query event in turn.

3.7.1 Query Started

This event is sent when the query is started on a node. It is sent using a **queryStarted** event element. This element does not take any attributes.

Listing 75: Query Started

```
<message from='concentrator@example.org'
                to='client@example.org/client'>
                <queryProgress xmlns='urn:xmpp:iot:concentrators'
                    sourceId='MeteringGroups' nodeId='Apartment 1-1'
                    cacheType='Apartment' queryId='Query0001'>
                  <queryStarted/>
                  ...
                </queryProgress>
```

```
            </message>
```

3.7.2 Query Done

This event is sent when the query is done on a node. It is sent using a **queryDone** event element. This element does not take any attributes.

Listing 76: Query Done

```
<message from='concentrator@example.org'
                to='client@example.org/client'>
                <queryProgress xmlns='urn:xmpp:iot:concentrators'
                    sourceId='MeteringGroups' nodeId='Apartment 1-1'
                    cacheType='Apartment' queryId='Query0001'>
                  ...
                  <queryDone/>
                </queryProgress>
              </message>
```

3.7.3 Query Aborted

This event is sent when the query has been aborted on a node. It is sent using a **queryAborted** event element. This element does not take any attributes.

Listing 77: Query Aborted

```
<message from='concentrator@example.org'
                to='client@example.org/client'>
                <queryProgress xmlns='urn:xmpp:iot:concentrators'
                    sourceId='MeteringGroups' nodeId='Apartment 1-1'
                    cacheType='Apartment' queryId='Query0001'>
                  ...
                  <queryAborted/>
                </queryProgress>
              </message>
```

3.7.4 New Table

A query result can consist of zero or more data tables. Each data table consists of a set of columns and records containing value items for the these columns. A new table is identified by a query event named **newTable**. a **newTable** event contains a sequence of **column** events each identifying a column in the table.

Listing 78: New Table

```
<message from='concentrator@example.org'
                to='client@example.org/client'>
                <queryProgress xmlns='urn:xmpp:iot:concentrators'
                    sourceId='MeteringGroups' nodeId='Apartment 1-1'
                    cacheType='Apartment' queryId='Query0001'>
                  ...
                  <newTable tableId='table1' tableName='Events'>
                    <column columnId='timestamp' header='Timestamp'>
                      <column columnId='message' header='Message'>
                      </newTable>
                      ...
                    </queryProgress>
                  </message>
```

For more information about column definitions, see Table Column definitions in query results.

3.7.5 New Records

A set of records in an open table is reported using the **newRecords** query event.

Listing 79: New Records

```
<message from='concentrator@example.org'
                    to='client@example.org/client'>
                    <queryProgress xmlns='urn:xmpp:iot:concentrators'
                        sourceId='MeteringGroups' nodeId='Apartment
                        1-1' cacheType='Apartment' queryId='Query0001'
                        >
                      ...
                      <newRecords tableId='table1'>
                        <record>
                          <dateTime>2013-04-24T17:43:15</dateTime>
                          <string>Harassed</string>
                        </record>
                        ...
                      </newRecords>
                      ...
                    </queryProgress>
                  </message>
```

For more information about records, see Record Item definitions in query results.

3.7.6 Table Done

This event is sent when the query is done with a table. It is sent using a **tableDone** event element. The query must not send any more records to a table that has been closed using this

query event.

Listing 80: Table Done

```
<message from='concentrator@example.org'
                    to='client@example.org/client'>
                    <queryProgress xmlns='urn:xmpp:iot:concentrators'
                        sourceId='MeteringGroups' nodeId='Apartment␣
                        1-1' cacheType='Apartment' queryId='Query0001'
                        >
                      ...
                      <tableDone tableId='table1'/>
                      ...
                    </queryProgress>
                  </message>
```

3.7.7 New Object

Some queries may want to return other types of data than tables, like images, graphs, etc. These may be combined with table information or not. A query returns such an object using the **newObject** query event. Each object is MIME encoded, and the MIME Type is sent in the **contentType** attribute. The object itself is Base-64 encoded.

Listing 81: New Object

```
<message from='concentrator@example.org'
                    to='client@example.org/client'>
                    <queryProgress xmlns='urn:xmpp:iot:concentrators'
                        sourceId='MeteringGroups' nodeId='Apartment␣
                        1-1' cacheType='Apartment' queryId='Query0001'
                        >
                      ...
                      <newObject contentType='image/png'>...</
                         newObject>
                      ...
                    </queryProgress>
                  </message>
```

3.7.8 Query Message

During the processing of a query, the query might want to report a message of some kind. This is done using the **queryMessage** query event.

Listing 82: Query Message

```
<message from='concentrator@example.org'
```

```
                    to='client@example.org/client'>
                    <queryProgress xmlns='urn:xmpp:iot:concentrators'
                        sourceId='MeteringGroups' nodeId='Apartment␣
                        1-1' cacheType='Apartment' queryId='Query0001'
                        >
                      ...
                      <queryMessage type='Information' level='Minor'>
                          30 events found.</queryMessage>
                      ...
                    </queryProgress>
                  </message>
```

For more information about query messages, see Message definitions in query results.

3.7.9 Title

Enables the executing process of the query to set a custom title for the result. Note that the result only has one title.

Listing 83: Title

```
<message from='concentrator@example.org'
                    to='client@example.org/client'>
                    <queryProgress xmlns='urn:xmpp:iot:concentrators'
                        sourceId='MeteringGroups' nodeId='Apartment␣
                        1-1' cacheType='Apartment' queryId='Query0001'
                        >
                      ...
                      <title name='Current␣state'/>
                      ...
                    </queryProgress>
                  </message>
```

3.7.10 Status

Permits the executing process to report the current status. This can change a lot during execution and values should not be persisted, like query messages. Instead, they can be displayed in a status bar, just to give the end-user feedback of the status of a long-running query.

Listing 84: Status

```
<message from='concentrator@example.org'
                    to='client@example.org/client'>
```

```
                    <queryProgress xmlns='urn:xmpp:iot:concentrators'
                        sourceId='MeteringGroups' nodeId='Apartment␣
                        1-1' cacheType='Apartment' queryId='Query0001'
                        >
                      ...
                      <status message='Processing.␣67%␣done.'/>
                      ...
                    </queryProgress>
                  </message>
```

3.7.11 Starting a section or subsection

Starts a new section in the query result. In large query results, sections can be used to format the result using an intuitive disposition in a visually appealing way. Nested sections should be interepreted as sub-sections. Tables and objects started or reported within a section belong in that section, and should be listed sequentially in the order they are reported to the client.

Listing 85: Starting a section or subsection

```
<message from='concentrator@example.org'
                    to='client@example.org/client'>
                    <queryProgress xmlns='urn:xmpp:iot:concentrators'
                        sourceId='MeteringGroups' nodeId='Apartment␣
                        1-1' cacheType='Apartment' queryId='Query0001'
                        >
                      ...
                      <beginSection header='1.␣Stockholm'/>
                      ...
                      <beginSection header='1.1.␣Solna'/>
                      ...
                    </queryProgress>
                  </message>
```

Note that the use of sections in a query result is optional. Tables and objects received outside the scope of sections should be placed in a default section by the client.

3.7.12 Closing a section or subsection

Closes the current section or sub-section. Tables inside closed sections can still be open and receive fields. Tables are closed using the **tableDone** query event.

Listing 86: Starting a section or subsection

```
<message from='concentrator@example.org'
                    to='client@example.org/client'>
```

```
                <queryProgress xmlns='urn:xmpp:iot:concentrators'
                    sourceId='MeteringGroups' nodeId='Apartment 1-1'
                     cacheType='Apartment' queryId='Query0001'>
                  ...
                  <endSection/>
                  ...
                  <endSection/>
                  ...
                </queryProgress>
              </message>
```

3.8 Data Source Events

Events are asynchronous messages sent from the concentrator to devices registered with the concentrator and having subscribed to events from the concentrator. Event subscription can be activated using the **subscribe** command. You can also retrieve historical events using the **subscribe** command.
Each event is sent from the concentrator to each subscriber using a **message** stanza. Each such **message** stanza may contain multiple events. The following subsections list the different types of events that can be sent. Even though these examples may list only one element in a message stanza, this is not a limitation. Different types of events may also be mixed within the message stanza.
Event subscriptions only last for as long as the client and concentrator both maintains presence. The concentrator must not persist event notification subscriptions, and if it goes offline and back online, or if the client goes offline or online again for any reason, the event subscription is removed.
Node events may contain parameters or status messages. This depends on how the subscription was made. The **parameters** and **messages** attributes in the **subscribe** command determine if node parameters and/or status messages should be sent in event messages.

3.8.1 Node added event

The **nodeAdded** event is sent when a node has been added to a data source which the client subscribes to. The following example shows how such an event message may look like:

Listing 87: Node added event, without parameters

```
<message from='concentrator@example.org'
                 to='client@example.org/client'>
                 <nodeAdded xmlns='urn:xmpp:iot:concentrators'
                    sourceId='MeteringTopology' nodeId='Node1'
                    nodeType='Namespace.NodeType1'
                   cacheType='Node' state='None' hasChildren='false
                      ' isReadable='true' isControllable='true'
                      hasCommands='true'
```

```
                    parentId='Root' lastChanged='2013-03-19T17:58:01
                        '/>
                </message>
```

Listing 88: Node added event, with parameters

```
<message from='concentrator@example.org'
                    to='client@example.org/client'>
                    <nodeAdded xmlns='urn:xmpp:iot:concentrators'
                        sourceId='MeteringTopology' nodeId='Node1'
                        nodeType='Namespace.NodeType1'
                        cacheType='Node' state='None' hasChildren='false
                        ' isReadable='true' isControllable='true'
                        hasCommands='true'
                        parentId='Root' lastChanged='2013-03-19T17:58:01
                        '>
                        <string id='id' name='Node ID' value='Node1'/>
                        <string id='type' name='Node Type' value='
                            Watchamacallit Temperature Sensor v1.2'/>
                        <string id='sn' name='Serial Number' value='
                            123456'/>
                        <string id='class' name='Node Class' value='
                            Temperature'/>
                        <string id='meterLoc' name='Meter Location'
                            value='P123502-2'/>
                        <int id='addr' name='Address' value='123'/>
                        <double id='lat' name='Latitude' value='12.345'/
                            >
                        <double id='long' name='Longitude' value='123.45
                            '/>
                    </nodeAdded>
                </message>
```

Note: Moving a node in a data source from one parent to another, possibly between different data sources, is performed by first removing it from the current source (sending a **nodeRemoved** event) and then adding it to the new data source (sending a **nodeAdded** event), in that order.
If a node is added to a parent where order of nodes is important, the attributes **afterNodeId** and also optionally **afterNodeCacheType** used to identify the node after which the new node should be inserted.

Listing 89: Ordered Node added event, without parameters

```
<message from='concentrator@example.org'
                    to='client@example.org/client'>
                    <nodeAdded xmlns='urn:xmpp:iot:concentrators'
                        sourceId='MeteringTopology' nodeId='Node2'
                        nodeType='Namespace.NodeType1'
```

```
                    cacheType='Node' state='None' hasChildren='false
                        ' isReadable='true' isControllable='true'
                        hasCommands='true'
                    parentId='Root' lastChanged='2013-03-19T17:58:01
                        ' afterNodeId='Node1' afterNodeCacheType='
                        Node'/>
                </message>
```

3.8.2 Node removed

The **nodeRemoved** event is sent when a node in a data source has been removed or destroyed. It does not include status message or parameter information about the node, only the reference to the node. The following example shows how such an event message may look like:

Listing 90: Node removed

```
<message from='concentrator@example.org'
                    to='client@example.org/client'>
                    <nodeRemoved xmlns='urn:xmpp:iot:concentrators'
                        sourceId='MeteringTopology' nodeId='Node1'/>
                </message>
```

Note: Moving a node in a data source from one parent to another, possibly between different data sources, is performed by first removing it from the current source (sending a **nodeRemoved** event) and then adding it to the new data source (sending a **nodeAdded** event), in that order.

3.8.3 Node updated

The **nodeUpdated** event is sent when a node has been updated in a data source which the client subscribes to. The following example shows how such an event message may look like:

Listing 91: Node updated event, without parameters

```
<message from='concentrator@example.org'
                    to='client@example.org/client'>
                    <nodeUpdated xmlns='urn:xmpp:iot:concentrators'
                        sourceId='MeteringTopology' nodeId='Node1'
                        nodeType='Namespace.NodeType1'
                        cacheType='Node' state='None' hasChildren='false
                        ' isReadable='true' isControllable='true'
                        hasCommands='true'
                        parentId='Root' lastChanged='2013-03-19T17:58:01
                        '/>
                </message>
```

Listing 92: Node updated event, with parameters

```
<message from='concentrator@example.org'
                    to='client@example.org/client'>
                    <nodeUpdated xmlns='urn:xmpp:iot:concentrators'
                        sourceId='MeteringTopology' nodeId='Node1'
                        nodeType='Namespace.NodeType1'
                      cacheType='Node' state='None' hasChildren='false
                          ' isReadable='true' isControllable='true'
                          hasCommands='true'
                      parentId='Root' lastChanged='2013-03-19T17:58:01
                          '>
                      <string id='id' name='Node ID' value='Node1'/>
                      <string id='type' name='Node Type' value='
                          Watchamacallit Temperature Sensor v1.2'/>
                      <string id='sn' name='Serial Number' value='
                          123456'/>
                      <string id='class' name='Node Class' value='
                          Temperature'/>
                      <string id='meterLoc' name='Meter Location'
                          value='P123502-2'/>
                      <int id='addr' name='Address' value='123'/>
                      <double id='lat' name='Latitude' value='12.345'/
                          >
                      <double id='long' name='Longitude' value='123.45
                          '/>
                    </nodeUpdated>
                  </message>
```

The event has an optional attribute named **oldId**. It is set when the node has been renamed. It must be used, if available, to identify what node has been updated, as the following example shows:

Listing 93: Node renamed event, without parameters

```
<message from='concentrator@example.org'
                    to='client@example.org/client'>
                    <nodeUpdated xmlns='urn:xmpp:iot:concentrators'
                        sourceId='MeteringTopology' oldId='OldNode1'
                        nodeId='Node1'
                      nodeType='Namespace.NodeType1' cacheType='Node'
                          state='None' hasChildren='false' isReadable=
                          'true'
                      isControllable='true' hasCommands='true'
                          parentId='Root' lastChanged='2013-03-19
                          T17:58:01'/>
                  </message>
```

Listing 94: Node renamed event, with parameters

```
<message from='concentrator@example.org'
                    to='client@example.org/client'>
                    <nodeUpdated xmlns='urn:xmpp:iot:concentrators'
                        sourceId='MeteringTopology' oldId='OldNode1'
                        nodeId='Node1'
                      nodeType='Namespace.NodeType1' cacheType='Node'
                        state='None' hasChildren='false' isReadable=
                        'true'
                      isControllable='true' hasCommands='true'
                        parentId='Root' lastChanged='2013-03-19
                        T17:58:01'>
                      <string id='id' name='Node ID' value='Node1'/>
                      <string id='type' name='Node Type' value='
                        Watchamacallit Temperature Sensor v1.2'/>
                      <string id='sn' name='Serial Number' value='
                        123456'/>
                      <string id='class' name='Node Class' value='
                        Temperature'/>
                      <string id='meterLoc' name='Meter Location'
                        value='P123502-2'/>
                      <int id='addr' name='Address' value='123'/>
                      <double id='lat' name='Latitude' value='12.345'/
                        >
                      <double id='long' name='Longitude' value='123.45
                        '/>
                    </nodeUpdated>
                  </message>
```

Note: If only the status has changed, and no other parameters have changed, the nodeStatusChanged event could be sent instead, for somewhat improved performance in this case.

3.8.4 Node status changed

The **nodeStatusChanged** event is sent when the state of the node has changed, but no other parameters have been changed. If the client has subscribed to state messages, these will also be included in the event.

Listing 95: Node status changed event, without messages

```
<message from='concentrator@example.org'
                    to='client@example.org/client'>
                    <nodeStatusChanged xmlns='
                        urn:xmpp:iot:concentrators' sourceId='
                        MeteringTopology' nodeId='Node1' cacheType='
                        Node' state='ErrorUnsigned'/>
                  </message>
```

Listing 96: Node status changed event, with messages

```
<message from='concentrator@example.org'
                    to='client@example.org/client'>
                    <nodeStatusChanged xmlns='
                        urn:xmpp:iot:concentrators' sourceId='
                        MeteringTopology' nodeId='Node1' cacheType='
                        Node' state='ErrorUnsigned'>
                      <message timestamp='2013-03-22T12:49:34' type='
                          Error'>Sensor does not respond to read-out
                          requests.</message>
                    </nodeStatusChanged>
                  </message>
```

3.8.5 Node moved up

The **nodeMovedUp** event is sent when a node in a data source has been moved up among its siblings. It does not include status message or parameter information about the node, only the reference to the node. The following example shows how such an event message may look like:

Listing 97: Node moved up

```
<message from='concentrator@example.org'
                    to='client@example.org/client'>
                    <nodeMovedUp xmlns='urn:xmpp:iot:concentrators'
                        sourceId='MeteringTopology' nodeId='Node1'/>
                  </message>
```

Note: If issuing a **moveNodeUp** command on a node that is already first among its siblings, the node is not moved and the concentrator must not send a **nodeMovedUp** event. However, if the client receives such an event pointing to a node that is already first among its siblings, it should not move the node.

3.8.6 Node moved down

The **nodeMovedDown** event is sent when a node in a data source has been moved down among its siblings. It does not include status message or parameter information about the node, only the reference to the node. The following example shows how such an event message may look like:

Listing 98: Node moved down

```
<message from='concentrator@example.org'
                    to='client@example.org/client'>
                    <nodeMovedDown xmlns='urn:xmpp:iot:concentrators'
                        sourceId='MeteringTopology' nodeId='Node1'/>
                  </message>
```

Note: If issuing a **moveNodeDown** command on a node that is already last among its siblings, the node is not moved and the concentrator must not send a **nodeMovedDown** event. However, if the client receives such an event pointing to a node that is already last among its siblings, it should not move the node.

3.9 Field Databases

A concentrator can often store data from sensors locally (or remotely but controlled locally). Storage is done in **field databases**. A concentrator with only communicative abilities will publish zero such field databases (and support for none of the field database commands), while a pure metering database will publish one or many field databases, but none of the nodes available in any of the different data sources are readable or writable. The nodes in this latter case only acts as placeholders for the data that the concentrator is storing or controlling.
The following subsections lists different use cases relating to field databases and how to use them.

3.9.1 Get All Databases

The client can retrieve the list of available databases on the concentrator using the **getDatabases** command, as is shown in the following example:

Listing 99: Get All Databases

```
<iq type='get'
                    from='client@example.org/client'
                    to='concentrator@example.org'
                    id='64'>
                    <getDatabases xmlns='urn:xmpp:iot:concentrators'
                        xml:lang='en'/>
                  </iq>

                  <iq type='result'
                    from='concentrator@example.org'
                    to='client@example.org/client'
                    id='64'>
                    <getDatabasesResponse xmlns='
                        urn:xmpp:iot:concentrators' result='OK'>
                      <database databaseId='LocalDB' name='Local␣
                          Database'/>
                      <database databaseId='LocalDBTest' name='Local␣
                          Test␣Database'/>
                      <database databaseId='LocalDBProduction' name='
                          Local␣Production␣Database'/>
                      <database databaseId='RemoteDB' name='Remote␣
                          Database'/>
```

```
    </getDatabasesResponse>
  </iq>
```

3.9.2 Get Database Readout Parameters

Before reading data from a database can begin, a parameter form containing database specific search parameters needs to be fetched from the server. This is done using the **getDatabaseReadoutParameters** command, as is shown in the following example:

Listing 100: Get Database Readout Parameters

```
<iq type='get'
    from='client@example.org/client'
    to='concentrator@example.org'
    id='65'>
    <getDatabaseReadoutParameters xmlns='
        urn:xmpp:iot:concentrators' xml:lang='en'
        databaseId='LocalDB'/>
  </iq>

  <iq type='result'
    from='concentrator@example.org'
    to='client@example.org/client'
    id='65'>
    <getDatabaseReadoutParametersResponse xmlns='
        urn:xmpp:iot:concentrators' result='OK'>
      <x type='form'
        xmlns='jabber:x:data'
        xmlns:xdv='http://jabber.org/protocol/xdata-
            validate'
        xmlns:xdl='http://jabber.org/protocol/xdata-
            layout'
        xmlns:xdd='urn:xmpp:xdata:dynamic'>
        <title>Database read-out</title>
        <xdl:page label='Time'>
          <xdl:fieldref var='from'/>
          <xdl:fieldref var='to'/>
        </xdl:page>
        <xdl:page label='Fields'>
          <xdl:fieldref var='fields'/>
        </xdl:page>
        <xdl:page label='Types'>
          <xdl:fieldref var='momentary'/>
          <xdl:fieldref var='peak'/>
          <xdl:fieldref var='status'/>
          <xdl:fieldref var='computed'/>
          <xdl:fieldref var='identity'/>
          <xdl:fieldref var='historySecond'/>
```

```
      <xdl:fieldref var='historyMinute'/>
      <xdl:fieldref var='historyHour'/>
      <xdl:fieldref var='historyDay'/>
      <xdl:fieldref var='historyWeek'/>
      <xdl:fieldref var='historyMonth'/>
      <xdl:fieldref var='historyQuarter'/>
      <xdl:fieldref var='historyYear'/>
      <xdl:fieldref var='historyOther'/>
    </xdl:page>
    <xdl:page label='Status'>
      <xdl:fieldref var='missing'/>
      <xdl:fieldref var='automaticEstimates'/>
      <xdl:fieldref var='manualEstimates'/>
      <xdl:fieldref var='manualReadout'/>
      <xdl:fieldref var='automaticReadout'/>
      <xdl:fieldref var='timeOffset'/>
      <xdl:fieldref var='warning'/>
      <xdl:fieldref var='error'/>
      <xdl:fieldref var='signed'/>
      <xdl:fieldref var='invoiced'/>
      <xdl:fieldref var='invoicedConfirmed'/>
      <xdl:fieldref var='endOfSeries'/>
      <xdl:fieldref var='powerFailure'/>
      <xdl:fieldref var='statusLogicMode'/>
    </xdl:page>
    <field var='xdd_session' type='hidden'>
      <value>C20D4F0C-8F8A-490F-8188-53017DF2B71A<
        /value>
    </field>
    <field var='from' type='text-single' label='
      From:'>
      <required/>
      <xdv:validate datatype='xs:dateTime'>
        <xdv:basic/>
      </xdv:validate>
      <desc>Read values not older that this
        timestamp.</desc>
      <value></value>
    </field>
    <field var='to' type='text-single' label='To:'
      >
      <required/>
      <xdv:validate datatype='xs:dateTime'>
        <xdv:basic/>
      </xdv:validate>
      <desc>Read values not newer that this
        timestamp.</desc>
      <value></value>
    </field>
```

```
<field var='fields' type='text-multi' label='
    Fields to include: (Use * as wildcards)'>
  <desc>If specified, only the following
     fields (include wildcards) will be read
     out.</desc>
  <value></value>
</field>
<field var='momentary' type='boolean' label='
    Momentary Values'>
  <desc>If checked, momentary values will be
     read.</desc>
  <value>false</value>
</field>
<field var='peak' type='boolean' label='Peak
    Values'>
  <desc>If checked, peak values will be read.<
     /desc>
  <value>false</value>
</field>
<field var='status' type='boolean' label='
    Status Values'>
  <desc>If checked, status values will be read
     .</desc>
  <value>false</value>
</field>
<field var='computed' type='boolean' label='
    Computed Values'>
  <desc>If checked, computed values will be
     read.</desc>
  <value>false</value>
</field>
<field var='identity' type='boolean' label='
    Identity Values'>
  <desc>If checked, identity values will be
     read.</desc>
  <value>false</value>
</field>
<field var='historySecond' type='boolean'
    label='Historical Values (second)'>
  <desc>If checked, historical values (second)
      will be read.</desc>
  <value>false</value>
</field>
<field var='historyMinute' type='boolean'
    label='Historical Values (minute)'>
  <desc>If checked, historical values (minute)
      will be read.</desc>
  <value>false</value>
</field>
```

```
                    <field var='historyHour' type='boolean' label=
                        'Historical Values (hour)'>
                      <desc>If checked, historical values (hour)
                          will be read.</desc>
                      <value>false</value>
                    </field>
                    <field var='historyDay' type='boolean' label='
                        Historical Values (day)'>
                      <desc>If checked, historical values (day)
                          will be read.</desc>
                      <value>false</value>
                    </field>
                    <field var='historyWeek' type='boolean' label=
                        'Historical Values (week)'>
                      <desc>If checked, historical values (week)
                          will be read.</desc>
                      <value>false</value>
                    </field>
                    <field var='historyMonth' type='boolean' label
                        ='Historical Values (month)'>
                      <desc>If checked, historical values (month)
                          will be read.</desc>
                      <value>false</value>
                    </field>
                    <field var='historyQuarter' type='boolean'
                        label='Historical Values (quarter)'>
                      <desc>If checked, historical values (quarter
                          ) will be read.</desc>
                      <value>false</value>
                    </field>
                    <field var='historyYear' type='boolean' label=
                        'Historical Values (year)'>
                      <desc>If checked, historical values (year)
                          will be read.</desc>
                      <value>false</value>
                    </field>
                    <field var='historyOther' type='boolean' label
                        ='Historical Values (other)'>
                      <desc>If checked, historical values (other)
                          will be read.</desc>
                      <value>false</value>
                    </field>
                    <field var='missing' type='list-single' label=
                        'Missing Values:'>
                      <desc>Determines how this flag should be
                          treated during readout.</desc>
                      <value>Ignore</value>
                      <option label='Ignore'><value>Ignore this
                          flag.</value></option>
```

```
      <option label='Required'><value>Only read
          values with this flag set.</value></
          option>
      <option label='Prohibit'><value>Only read
          values with this flag cleared.</value></
          option>
    </field>
    <field var='automaticEstimates' type='list-
        single' label='Automatic␣Estimates:'>
      <desc>Determines how this flag should be
          treated during readout.</desc>
      <value>Ignore</value>
      <option label='Ignore'><value>Ignore this
          flag.</value></option>
      <option label='Required'><value>Only read
          values with this flag set.</value></
          option>
      <option label='Prohibit'><value>Only read
          values with this flag cleared.</value></
          option>
    </field>
    <field var='manualEstimates' type='list-single
        ' label='Manual␣Estimates:'>
      <desc>Determines how this flag should be
          treated during readout.</desc>
      <value>Ignore</value>
      <option label='Ignore'><value>Ignore this
          flag.</value></option>
      <option label='Required'><value>Only read
          values with this flag set.</value></
          option>
      <option label='Prohibit'><value>Only read
          values with this flag cleared.</value></
          option>
    </field>
    <field var='manualReadout' type='list-single'
        label='Manual␣Readout:'>
      <desc>Determines how this flag should be
          treated during readout.</desc>
      <value>Ignore</value>
      <option label='Ignore'><value>Ignore this
          flag.</value></option>
      <option label='Required'><value>Only read
          values with this flag set.</value></
          option>
      <option label='Prohibit'><value>Only read
          values with this flag cleared.</value></
          option>
    </field>
```

```
<field var='automaticReadout' type='list-
    single' label='Automatic Readout:'>
  <desc>Determines how this flag should be
      treated during readout.</desc>
  <value>Ignore</value>
  <option label='Ignore'><value>Ignore this
      flag.</value></option>
  <option label='Required'><value>Only read
      values with this flag set.</value></
      option>
  <option label='Prohibit'><value>Only read
      values with this flag cleared.</value></
      option>
</field>
<field var='timeOffset' type='list-single'
    label='Time Offset:'>
  <desc>Determines how this flag should be
      treated during readout.</desc>
  <value>Ignore</value>
  <option label='Ignore'><value>Ignore this
      flag.</value></option>
  <option label='Required'><value>Only read
      values with this flag set.</value></
      option>
  <option label='Prohibit'><value>Only read
      values with this flag cleared.</value></
      option>
</field>
<field var='warning' type='list-single' label=
    'Warning:'>
  <desc>Determines how this flag should be
      treated during readout.</desc>
  <value>Ignore</value>
  <option label='Ignore'><value>Ignore this
      flag.</value></option>
  <option label='Required'><value>Only read
      values with this flag set.</value></
      option>
  <option label='Prohibit'><value>Only read
      values with this flag cleared.</value></
      option>
</field>
<field var='error' type='list-single' label='
    Error:'>
  <desc>Determines how this flag should be
      treated during readout.</desc>
  <value>Ignore</value>
  <option label='Ignore'><value>Ignore this
      flag.</value></option>
```

```
    <option label='Required'><value>Only read
        values with this flag set.</value></
        option>
    <option label='Prohibit'><value>Only read
        values with this flag cleared.</value></
        option>
</field>
<field var='signed' type='list-single' label='
    Signed:'>
    <desc>Determines how this flag should be
        treated during readout.</desc>
    <value>Ignore</value>
    <option label='Ignore'><value>Ignore this
        flag.</value></option>
    <option label='Required'><value>Only read
        values with this flag set.</value></
        option>
    <option label='Prohibit'><value>Only read
        values with this flag cleared.</value></
        option>
</field>
<field var='invoiced' type='list-single' label
    ='Invoiced:'>
    <desc>Determines how this flag should be
        treated during readout.</desc>
    <value>Ignore</value>
    <option label='Ignore'><value>Ignore this
        flag.</value></option>
    <option label='Required'><value>Only read
        values with this flag set.</value></
        option>
    <option label='Prohibit'><value>Only read
        values with this flag cleared.</value></
        option>
</field>
<field var='invoicedConfirmed' type='list-
    single' label='Invoiced␣and␣Confirmed:'>
    <desc>Determines how this flag should be
        treated during readout.</desc>
    <value>Ignore</value>
    <option label='Ignore'><value>Ignore this
        flag.</value></option>
    <option label='Required'><value>Only read
        values with this flag set.</value></
        option>
    <option label='Prohibit'><value>Only read
        values with this flag cleared.</value></
        option>
</field>
```

```
                    <field var='endOfSeries' type='list-single'
                        label='End of Series:'>
                      <desc>Determines how this flag should be
                          treated during readout.</desc>
                      <value>Ignore</value>
                      <option label='Ignore'><value>Ignore this
                          flag.</value></option>
                      <option label='Required'><value>Only read
                          values with this flag set.</value></
                          option>
                      <option label='Prohibit'><value>Only read
                          values with this flag cleared.</value></
                          option>
                    </field>
                    <field var='powerFailure' type='list-single'
                        label='Power Failure:'>
                      <desc>Determines how this flag should be
                          treated during readout.</desc>
                      <value>Ignore</value>
                      <option label='Ignore'><value>Ignore this
                          flag.</value></option>
                      <option label='Required'><value>Only read
                          values with this flag set.</value></
                          option>
                      <option label='Prohibit'><value>Only read
                          values with this flag cleared.</value></
                          option>
                    </field>
                    <field var='statusLogicMode' type='list-single
                        ' label='Combination Logic:'>
                      <desc>This parameter specifies how multiple
                          status flags are to be treated in the
                          readout.</desc>
                      <value>Any</value>
                      <option label='Any'><value>Any of the
                          conditions above are met.</value></
                          option>
                      <option label='All'><value>All of the
                          conditions above are met.</value></
                          option>
                    </field>
                  </x>
                </getDatabaseReadoutParametersResponse>
              </iq>
```

3.9.3 Start Database Readout

Once the parameter form has been filled out, a database readout can be started using the **startDatabaseReadout** command. Data read from the concentrator will follow the same flow of sensor data as that described in Internet of Things - Sensor Data (XEP-0323) [5]. The only difference is that the read-out is started with the **startDatabaseReadout** command instead of the **req** command of the sensor data namespace.
The following diagram shows the flow of messages when requesting a readout from a database:

The following example shows a **startDatabaseReadout** request:

Listing 101: Start Database Readout

```
<iq type='set'
                    from='client@example.org/client'
                    to='concentrator@example.org'
                    id='66'>
                    <startDatabaseReadout xmlns='
                        urn:xmpp:iot:concentrators' xml:lang='en'
                        databaseId='LocalDB' seqnr='1' userToken='
                        12392748927492834'>
                      <node sourceId='MeteringTopology' nodeId='Node1'
                          />
                      <node sourceId='MeteringTopology' nodeId='Node2'
                          />
                      <node sourceId='MeteringTopology' nodeId='Node3'
                          />
                      <x type='submit' xmlns='jabber:x:data'>
                        <field var='xdd_session' type='hidden'>
                          <value>C20D4F0C-8F8A-490F-8188-53017DF2B71A<
                              /value>
                        </field>
                        <field var='from' type='text-single'>
                          <value>2013-03-22T00:00:00</value>
                        </field>
                        <field var='to' type='text-single'>
                          <value>2013-03-23T00:00:00</value>
                        </field>
                        <field var='fields' type='text-multi'>
                          <value>Temperature</value>
                          <value>Pressure</value>
                          <value>Power</value>
                        </field>
                        <field var='momentary' type='boolean'>
```

[5] XEP-0323: Internet of Things - Sensor Data <https://xmpp.org/extensions/xep-0323.html>.

```
      <value>true</value>
    </field>
    <field var='peak' type='boolean'>
      <value>false</value>
    </field>
    <field var='status' type='boolean'>
      <value>false</value>
    </field>
    <field var='computed' type='boolean'>
      <value>false</value>
    </field>
    <field var='identity' type='boolean'>
      <value>false</value>
    </field>
    <field var='historySecond' type='boolean'>
      <value>false</value>
    </field>
    <field var='historyMinute' type='boolean'>
      <value>false</value>
    </field>
    <field var='historyHour' type='boolean'>
      <value>false</value>
    </field>
    <field var='historyDay' type='boolean'>
      <value>false</value>
    </field>
    <field var='historyWeek' type='boolean'>
      <value>false</value>
    </field>
    <field var='historyMonth' type='boolean'>
      <value>false</value>
    </field>
    <field var='historyQuarter' type='boolean'>
      <value>false</value>
    </field>
    <field var='historyYear' type='boolean'>
      <value>false</value>
    </field>
    <field var='historyOther' type='boolean'>
      <value>false</value>
    </field>
    <field var='missing' type='list-single'>
      <value>Ignore</value>
    </field>
    <field var='automaticEstimates' type='list-
       single'>
      <value>Ignore</value>
    </field>
    <field var='manualEstimates' type='list-single
```

```
                    '>
                  <value>Ignore</value>
                </field>
                <field var='manualReadout' type='list-single'>
                  <value>Ignore</value>
                </field>
                <field var='automaticReadout' type='list-
                    single'>
                  <value>Ignore</value>
                </field>
                <field var='timeOffset' type='list-single'>
                  <value>Ignore</value>
                </field>
                <field var='warning' type='list-single'>
                  <value>Ignore</value>
                </field>
                <field var='error' type='list-single'>
                  <value>Ignore</value>
                </field>
                <field var='signed' type='list-single'>
                  <value>Ignore</value>
                </field>
                <field var='invoiced' type='list-single'>
                  <value>Ignore</value>
                </field>
                <field var='invoicedConfirmed' type='list-
                    single'>
                  <value>Ignore</value>
                </field>
                <field var='endOfSeries' type='list-single'>
                  <value>Ignore</value>
                </field>
                <field var='powerFailure' type='list-single'>
                  <value>Ignore</value>
                </field>
                <field var='statusLogicMode' type='list-single
                    '>
                  <value>Any</value>
                </field>
              </x>
            </startDatabaseReadout>
          </iq>

          <iq type='result'
            from='concentrator@example.org'
            to='client@example.org/client'
            id='66'>
            <startDatabaseReadoutResponse xmlns='
                urn:xmpp:iot:concentrators' result='OK'/>
```

```
                    </iq>

                    ... Readout messages follow.
```

3.9.4 Start Database Readout, Failure

The following example shows a response when the concentrator rejects the read-out request:

Listing 102: Start Database Readout, Failure

```
<iq type='set'
                      from='client@example.org/client'
                      to='concentrator@example.org'
                      id='67'>
                      <startDatabaseReadout xmlns='
                          urn:xmpp:iot:concentrators' xml:lang='en'
                          databaseId='LocalDB' seqnr='2' userToken='
                          12392748927492834'>
                        <node sourceId='MeteringTopology' nodeId='Node1'
                            />
                        <node sourceId='MeteringTopology' nodeId='Node2'
                            />
                        <node sourceId='MeteringTopology' nodeId='Node3'
                            />
                        <x type='submit' xmlns='jabber:x:data'>
                          ...
                        </x>
                      </startDatabaseReadout>
                    </iq>

                    <iq type='error'
                      from='concentrator@example.org'
                      to='client@example.org/client'
                      id='67'>
                      <startDatabaseReadoutResponse xmlns='
                          urn:xmpp:iot:concentrators' result='
                          InsufficientPrivileges'/>
                    </iq>
```

The error might also be a result of invalid input parameters being sent:

Listing 103: Start Database Readout, Failure 2

```
<iq type='set'
                      from='client@example.org/client'
                      to='concentrator@example.org'
                      id='68'>
```

```
                    <startDatabaseReadout xmlns='
                        urn:xmpp:iot:concentrators' xml:lang='en'
                        databaseId='LocalDB' seqnr='3' userToken='
                        12392748927492834'>
                      <node sourceId='MeteringTopology' nodeId='Node1'
                          />
                      <node sourceId='MeteringTopology' nodeId='Node2'
                          />
                      <node sourceId='MeteringTopology' nodeId='Node3'
                          />
                      <x type='submit' xmlns='jabber:x:data'>
                        ...
                        <field var='from' type='text-single'>
                          <value>2013-03-23T00:00:00</value>
                        </field>
                        <field var='to' type='text-single'>
                          <value>2013-03-22T00:00:00</value>
                        </field>
                        ...
                      </x>
                    </startDatabaseReadout>
                  </iq>

                  <iq type='error'
                    from='concentrator@example.org'
                    to='client@example.org/client'
                    id='68'>
                    <startDatabaseReadoutResponse xmlns='
                        urn:xmpp:iot:concentrators' result='FormError'
                        >
                      <error var='to'>The To timestamp cannot be older
                          than the From timestamp.</error>
                    </startDatabaseReadoutResponse>
                  </iq>
```

3.9.5 Cancelling a database read-out

A client can cancel a database read-out using the cancel read-out command, as described in Internet of Things - Sensor Data.

Listing 104: Cancelling a database read-out

```
<iq type='set'
                    from='client@example.org/client'
                    to='concentrator@example.org'
                    id='70'>
                    <startDatabaseReadout xmlns='
                        urn:xmpp:iot:concentrators' xml:lang='en'
```

```
            databaseId='LocalDB' seqnr='4' userToken='
            12392748927492834'>
          <node sourceId='MeteringTopology' nodeId='Node1'
              />
          <node sourceId='MeteringTopology' nodeId='Node2'
              />
          <node sourceId='MeteringTopology' nodeId='Node3'
              />
          <x type='submit' xmlns='jabber:x:data'>
            ...
          </x>
        </startDatabaseReadout>
      </iq>

      <iq type='result'
        from='concentrator@example.org'
        to='client@example.org/client'
        id='70'>
        <startDatabaseReadoutResponse xmlns='
            urn:xmpp:iot:concentrators' result='OK'/>
      </iq>

      ...

      <iq type='get'
        from='client@example.org/client'
        to='concentrator@example.org'
        id='71'>
        <cancel xmlns='urn:xmpp:iot:sensordata' seqnr='4'/
            >
      </iq>

      <iq type='result'
        from='concentrator@example.org'
        to='client@example.org/client'
        id='71'>
        <cancelled xmlns='urn:xmpp:iot:sensordata' seqnr='
            4'/>
      </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:concentrators" in response to Service Discovery (XEP-0030) [6] information requests.

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

Listing 105: Service discovery information request

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

Listing 106: Service discovery information response

```
<iq type='result'
                from='provisioning@example.org'
                to='device@example.org/device'
                id='disco1'>
                <query xmlns='http://jabber.org/protocol/disco#info'>
                  ...
                  <feature var='urn:xmpp:iot:concentrators'/>
                  ...
                </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) [7]. 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 Node Information

Several commands return basic node information. The following table lists possible fields with corresponding descriptions.

Attribute	Use	Default	Description
nodeId	required		The ID of the node in the data source.
displayName	optional		If provided, a string presentable to users. If localization is supported and a correct language attribute was provided, this string will be localized.

[7] XEP-0115: Entity Capabilities <https://xmpp.org/extensions/xep-0115.html>.

Attribute	Use	Default	Description
nodeType	optional		A string representing the type of the node.
localId	optional		If provided, an ID for the node, but unique locally between siblings.
logId	optional		If provided, an ID for the node, as it would appear or be used in system logs.
cacheType	optional		Used to uniquely identify the node in sources where the ID of the node is not sufficient. Example: In a spatial ordering different nodes may represent countries, regions, cities, areas, streets, buildings and apartments in the same source. However, the ID of each node would depend on what type of node it represents. It might be valid to have a region, city and/or area with the same ID. So, to these circumstances, a Cache Type of Country, Region, City, Area, Street, Building and Apartment can be used to allow for non-unique IDs within the source, as long as they are unique within the same cache type. ***
state	required		Current overall state of the node.

Attribute	Use	Default	Description
hasChildren	required		If the node has children or not.
childrenOrdered	optional	false	If the children of the node have an intrinsic order (true), or if the order is not important (false). If the order is not important, the client can present the order as it wishes, for example sorted on Node ID or any other parameter available on the child nodes. However, if the children have an intrinsic order, it's important for the client to refresh the child list when necessary whenever the order is unknown, such as after a nodeAdded event or a nodeRemoved event.
isReadable	optional	false	If the node can be read. (*)
isControllable	optional	false	If the node can be controlled. (**)
hasCommands	optional	false	If the node has registered commands or not.
parentId	optional		The node ID of the parent node. If not available, the node is considered a root node.
parentCacheType	optional		The Cache Type of the parent node.

Attribute	Use	Default	Description
lastChanged	optional		When the node was last changed. Can be used by clients to synchronize content between the concentrator and itself.

(*) See Internet of Things - Sensor Data for more information about how to read nodes.
(**) See Internet of Things - Control for more information about how to control nodes.
(***) Note that **cacheType** is optional. In some data sources it might be necessary to include a cache type to allow for IDs that are not unique within the source. In these cases, it concentrator must include the cacheType attribute for nodes in the corresponding data source, and clients must use the corresponding cacheType value when referencing the node. But if such a separation of IDs within a source is not required, the concentrator may ignore the cacheType attribute. In these instances clients may also ignore the cacheType attribute when referencing the nodes in the corresponding data source, or use the empty string as value for the cacheType value. The server must ignore empty cacheType values for data sources not using a cache type separation mechanism.

5.2 Parameter Types

Nodes can report parameters of different types. The following table lists available parameter types.

Element Name	Parameter Value
boolean	Boolean values. Can be true or false.
color	Color values. They are strings containing 6 case-insensitive hexadecimal digits of the form RRGGBB, where RR=Red component, GG=Green component, BB=Blue component.
dateTime	Date and time value, with possible time zone information.
double	double precision floating point value.
duration	A time duration value.
int	A 32-bit integer value.
long	A 64-bit integer value.
string	A string value.
time	A time value using a 24-hour clock.

5.3 Response Codes

All responses have a response code that the client is required to check. The following table lists possible response codes and their corresponding meanings.

Response Code	Description
OK	Request is OK and accepted.
NotFound	Corresponding object, node, etc., was not found.
InsufficientPrivileges	Insufficient privileges to perform the corresponding action or execute the corresponding command. Make sure to provide sufficient credentials in the call (user, service, device tokens) with sufficient privileges to perform the action/command.
Locked	The object/node is locked, and operation could not be performed or completed.
NotImplemented	Command is not implemented.
FormError	The form send to the concentrator contains errors. The response contains detailed error information.
OtherError	Another error occurred.

5.4 Unimplemented commands

It's important that the concentrator responds to unrecognized commands. This, to make sure future extensions to this document does not produce unexpected results in concentrators implementing a previous version, or in concentrators implementing only a subset of available commands.
When a concentrator receives an element that it does not understand, it must do as follows:

- If the element is received in a message stanza, the element must be ignored.
- If the element is received in an iq-result stanza, the element must be ignored.
- If the element is received in an iq-get or iq-set stanza, an iq-result stanza must be composed containing an element with the same name as the incoming element, but with **Response** appended to it, having a response code of **NotImplemented**.

The following example shows how unrecognized commands must be handled by the concentrator:

Listing 107: Unimplemented commands

```
<iq type='get'
                    from='client@example.org/client'
```

```
    to='concentrator@example.org'
    id='69'>
    <holabandola xmlns='urn:xmpp:iot:concentrators' ...>
      ...
    </holabandola>
  </iq>

  <iq type='error'
    from='concentrator@example.org'
    to='client@example.org/client'
    id='69'>
    <holabandolaResponse xmlns='
        urn:xmpp:iot:concentrators' result='
        NotImplemented'/>
  </iq>
```

5.5 Command attributes

Node commands have a set of attributes. The following table lists available attributes and what they mean.

Attribute	Use	Description
command	required	ID of the command. Used to identify the command.
name	required	A string that can be presented to an end-user. Should be localized if the request contained a language preference.
type	required	If the command is 'Simple' or 'Parameterized'.
sortCategory	optional	Should be used (if available) by clients to sort available node commands before presenting them to an end-user. Commands should be sorted by Sort Category, Sort Key and lastly by Name.
sortKey	optional	Should be used (if available) by clients to sort available node commands before presenting them to an end-user. Commands should be sorted by Sort Category, Sort Key and lastly by Name.

Attribute	Use	Description
confirmationString	optional	Should presented to clients (if available) before letting an end-user execute the command. A delete command might have a confirmationString saying 'Are you sure you want to delete the current item?' The confirmation string should be presented as a Yes/No[/Cancel] dialog.
failureString	optional	Could be presented to end-users (if available) if a command fails. It provides the client with an optionally localized string giving some context to the error message. A delete command might have a failureString saying 'Unable to delete the current item.'. The client could then add additional error information, if available, for instance from the response code.
successString	optional	Could be presented to end-users (if available) if a command is successfully executed. It provides the client with an optionally localized string giving some context to the message. A delete command might have a successString saying 'Current item successfully deleted.'.

5.6 Node States

Nodes can be in different states. Even though nodes are free to publish any amount of states as parameters, there are a set of states that are so crucial to the concept of a node state, that they are published using a separate attribute and separate events. These node states are as follows:

State	Description
None	The node has nothing special to report.

State	Description
Information	The node has informative events reported on it.
WarningSigned	The node has warnings reported on it. These warnings have been viewed by an operator.
WarningUnsigned	The node has warnings reported on it that have not been viewed by anybody.
ErrorSigned	The node has errors reported on it. These errors have been viewed by an operator.
ErrorUnsigned	The node has errors reported on it that have not been viewed by anybody.

5.7 Required Data Sources

The following table lists required data sources for concentrators (of sensors, actuators, meters, etc.) in sensor networks:

Data Source	Description
MeteringTopology	Data Source containing the topology of metering devices, including sensors, actuators, meter, infrastructure components, etc.

More information can be found in the [8].

5.8 Table Column definitions in query results

Table columns in query results can have the following attributes:

Attribute	Use	Description
columnId	Required	ID of column
header	Optional	If provided, a localized name to be displayed instead of the column ID.
alignment	Optional	Alignment of contents in the column. If provided, the client does not need to guess alignment from the type of data reported.

[8] XEP-xxxx: Interoperability Interoperability section

Attribute	Use	Description
nrDecimals	Optional	If floating point data is presented in the column and a fixed number of decimals is desired, this attribute is used to fix the number of decimals used.
dataSourceId	Optional	If contents of columns contains references to nodes in a data source, this attribute defines the data source ID.
cacheTypeName	Optional	If contents of columns contains references to nodes in a data source, this attribute defines the cache type.
fgColor	Optional	If presentation of this column should be done using a specific foreground color.
bgColor	Optional	If presentation of this column should be done using a specific background color.

5.9 Record Item definitions in query results

Records contain arrays of items, each item correspond to a specific column with the same ordinal index. Each item is specified together with its type. This, so receptors can integrate the result with type information if necessary. The following different type elements are available:

Element	Contents
boolean	A boolean value, corresponding to the xs:boolean data type.
color	A color value. These are strings containing 6 case-insensitive hexadecimal digits of the form RRGGBB, where RR=Red component, GG=Green component, BB=Blue component.
dateTime	A Date & Time value, corresponding to the xs:dateTime data type.
double	A floating-point value, corresponding to the xs:double data type.
duration	A duration value, corresponding to the xs:duration data type.
int	A 32-bit integer value, corresponding to the xs:int data type.
long	A 64-bit value, corresponding to the xs:long data type.
string	A string value, corresponding to the xs:string data type.
time	A time value, corresponding to the xs:time data type.
base64	A base-64 encoded object. The object is MIME encoded, and the attribute contentType contains the MIME Type which includes information on how to decode the value.

Element	Contents
void	Contains no information. This represents a NULL value.

5.10 Message definitions in query results

Messages in query results contain apart from a message text, also contain some classification on what the message is using two attributes: The **type** attribute gives a rough idea about the type of message and **level** the importance of the message.

Type	Description
Information	Information message
Warning	Message contains a warning. The situation could turn into an error if consideration or action is not taken.
Error	Message contains an error. The error message describes the error, and the error has been handled correctly by the device performing the query.
Exception	Message contains an exception message. An unforseen and perhaps an unhandled event has occurred during the execution of the query.

Level	Description
Minor	Minor event. Message describes common tasks.
Medium	Medium event. Message might describe a state change, something has been updated, or important information.
Major	Major event. Message might describe architectural change or vital information.

5.11 Single vs. batch commands

Many commands exist in single and batch versions. Large concentrators require efficient ways to manage sets of nodes simultaneously, while for small concentrators, it is sufficient to manage nodes one at a time.
A concentrator is not required to implement both sets of commands if not desired. However, it must report which commands it supports in the **getCapabilities** command. A caller can then adapt its calls based on what commands are supported.
If a caller wants to call the single item version of a command, but the concentrator only supports the batch version, it should call the batch version, but only include the single node in the list of nodes.
On the other hand, if the caller wants to perform an operation on a set of nodes, but the concentrator only supports the single item version of the command, it needs to manually

perform the operation on each node separately, unless failing the request is an option.

6 Internationalization Considerations

6.1 Localization

Localization of content can be performed if clients provide **xml:lang** attributes in commands made to the concentrator. If omitted, the **default language** will be used in responses. If provided, but the concentrator does not support localization, or the requested language, the **default language** will also be used.

6.2 Time Zones

Concentrators of larger sub-systems spanning multiple time-zones should specify all timestamps with time-zone information, so readers can perform comparisons of time information. Information read from a concentrator that lacks time-zone information should be considered to lie in the same time-zone as the reader, unless not explicitly configured with a time-zone.

6.3 Interoperability

For concentrators to be interoperable in sensor networks, they need to adhere to rules and guidelines described in the xep-0000-IoT-Interoperability document.

7 Security Considerations

7.1 Access rights

This document publishes a lot of commands with which to interact with a concentrator. If security and access rights is an issue, it might not be sufficient to allow all friends access to the system. There are many ways in which to restrict access to the contents of the concentrator. Following are some examples:

- The concentrator can restrict friendships to trusted friends, and then assign access rights internally to the approved contacts.
- The concentrator can use a provisioning server (see Internet of Things - Provisioning (XEP-0324) [9]) to delegate trust to a third party responsible for controlling who can get access to the concentrator (**isFriend** or **canAccess** commands), and what items can be viewed (**hasPrivilege** or **downloadPrivileges** commands).

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

- All requests to the concentrator can contain the optional attributes **deviceToken**, **serviceToken** and **userToken**. Clients making requests to the concentrator can use these attributes to forward information about who originated the action (**userToken**), what service is performing the action (**serviceToken**) or what device is performing the action (**deviceToken**). The concentrator can use this information to check with provisioning servers what access rights and user privileges exist before performing the action.

7.2 Integration with provisioning servers

The Internet of Things - Provisioning document describes how trust can be delegated to trusted provisioning servers that can be used to restrict access to and privileges in a network. If a concentrator has a trusted relationship with a provisioning server, external or internal, the provisioning server must be used to guarantee that the concentrator only allows access according to rules defined by the provisioning server. In order to do this, it's important that clients always provide available tokens (**userToken**, **serviceToken** and **deviceToken**) to the concentrator so that it can forward this information to the provisioning server.
The following subsections show different examples of how such an integration can be performed.

7.2.1 Restricting access to data source per contact

This section shows how a provisioning server can be used to restrict access to data sources to users allowed to access those data sources.

Notice the following:

- The user has a **userToken** it received during connection with the service. This user token should be included in all calls made by the user.
- The service has a **serviceToken** it received during service registration. This service token should be aggregated by all calls made by the service.
- The concentrators should cache all calls made to the provisioning server according to available cache rules.
- The conversion of a Data Source ID to a Privilege ID should be performed according to rules defined in xep-0000-IoT-Interoperability.

7.2.2 Restricting access to node properties per contact

This section shows how a provisioning server can be used to restrict access to node parameters that users allowed to access and edit.

- Not that privileges for all parameters needs to be checked in every call. And if lost from the cache, a new request needs to be made to the provisioning server.
- The conversion of a Parameter ID (and Page) to a Privilege ID should be performed according to rules defined in xep-0000-IoT-Interoperability.

7.2.3 Restricting access to node commands per contact

This section shows how a provisioning server can be used to restrict access to node commands to users allowed to access those commands.

In addition to rules noted above, also notice the following:

- Not that privileges for all parameters needs to be checked in every call. And if lost from the cache, a new request needs to be made to the provisioning server.
- The conversion of a Command ID to a Privilege ID should be performed according to rules defined in xep-0000-IoT-Interoperability.

8 IANA Considerations

This document requires no interaction with the Internet Assigned Numbers Authority (IANA) [10].

9 XMPP Registrar Considerations

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

[10] 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/>.

10 XML Schema

```
<?xml version='1.0' encoding='UTF-8'?>
<xs:schema
  xmlns:xs='http://www.w3.org/2001/XMLSchema'
  targetNamespace='urn:xmpp:iot:concentrators'
  xmlns='urn:xmpp:iot:concentrators'
  xmlns:sn='urn:xmpp:iot:sensordata'
  xmlns:xd="jabber:x:data"
  xmlns:xdv="http://jabber.org/protocol/xdata-validate"
  xmlns:xdl="http://jabber.org/protocol/xdata-layout"
  elementFormDefault='qualified'>

  <xs:import namespace='urn:xmpp:iot:sensordata'/>
  <xs:import namespace='jabber:x:data'/>
  <xs:import namespace='http://jabber.org/protocol/xdata-validate'/>
  <xs:import namespace='http://jabber.org/protocol/xdata-layout'/>

  <xs:element name='getCapabilities' type='TokenRequest'/>
  <xs:element name='getCapabilitiesResponse' type='StringArrayResponse
     '/>

  <xs:element name='getAllDataSources' type='TokenRequest'/>
  <xs:element name='getAllDataSourcesResponse' type='
     DataSourceArrayResponse'/>

  <xs:element name='getRootDataSources' type='TokenRequest'/>
  <xs:element name='getRootDataSourcesResponse' type='
     DataSourceArrayResponse'/>

  <xs:element name='getChildDataSources' type='SourceReferenceRequest'
     />
  <xs:element name='getChildDataSourcesResponse' type='
     DataSourceArrayResponse'/>

  <xs:element name='containsNode' type='NodeReferenceRequest'/>
  <xs:element name='containsNodeResponse'>
    <xs:complexType>
      <xs:simpleContent>
        <xs:extension base='xs:boolean'>
          <xs:attributeGroup ref='responseCode'/>
        </xs:extension>
      </xs:simpleContent>
    </xs:complexType>
  </xs:element>

  <xs:element name='containsNodes' type='NodeReferencesRequest'/>
  <xs:element name='containsNodesResponse' type='BooleanArrayResponse'
     />
```

```
<xs:element name='getNode' type='NodeReferenceParametersRequest'/>
<xs:element name='getNodeResponse'>
  <xs:complexType>
    <xs:complexContent>
      <xs:extension base='NodeInformation'>
        <xs:attributeGroup ref='responseCode'/>
      </xs:extension>
    </xs:complexContent>
  </xs:complexType>
</xs:element>

<xs:element name='getNodes' type='NodeReferenceParametersRequest'/>
<xs:element name='getNodesResponse' type='
    NodeInformationArrayResponse'/>

<xs:element name='getAllNodes' type='GetAllNodesRequest'/>
<xs:element name='getAllNodesResponse' type='
    NodeInformationArrayResponse'/>

<xs:element name='getNodeInheritance' type='NodeReferenceRequest'/>
<xs:element name='getNodeInheritanceResponse'>
  <xs:complexType>
    <xs:sequence>
      <xs:element name='baseClasses' type='StringArray' minOccurs='1
          ' maxOccurs='1'/>
      <xs:element name='interfaces' type='StringArray' minOccurs='0'
           maxOccurs='1'/>
    </xs:sequence>
  </xs:complexType>
</xs:element>

<xs:element name='getRootNodes' type='
    SourceReferenceParametersRequest'/>
<xs:element name='getRootNodesResponse' type='
    NodeInformationArrayResponse'/>

<xs:element name='getChildNodes' type='GetChildNodesRequest'/>
<xs:element name='getChildNodesResponse' type='
    NodeInformationArrayResponse'/>

<xs:element name='getIndices' type='SourceReferenceRequest'/>
<xs:element name='getIndicesResponse' type='StringArrayResponse'/>

<xs:element name='getNodesFromIndex' type='IndexReferenceRequest'/>
<xs:element name='getNodesFromIndexResponse' type='
    NodeInformationArrayResponse'/>
```

```
<xs:element name='getNodesFromIndices' type='IndexReferencesRequest'
    />
<xs:element name='getNodesFromIndicesResponse' type='
    NodeInformationArrayResponse'/>

<xs:element name='getAllIndexValues'>
  <xs:complexType>
    <xs:complexContent>
      <xs:extension base='SourceReferenceRequest'>
        <xs:attribute name='index' type='xs:string' use='required'/>
      </xs:extension>
    </xs:complexContent>
  </xs:complexType>
</xs:element>
<xs:element name='getAllIndexValuesResponse' type='
    StringArrayResponse'/>

<xs:element name='getNodeParametersForEdit' type='
    NodeReferenceRequest'/>
<xs:element name='getNodeParametersForEditResponse' type='
    ParameterFormResponse'/>

<xs:element name='setNodeParametersAfterEdit'>
  <xs:complexType>
    <xs:complexContent>
      <xs:extension base='NodeReferenceRequest'>
        <xs:sequence>
          <xs:element ref='xd:x' minOccurs="1" maxOccurs="1"/>
        </xs:sequence>
      </xs:extension>
    </xs:complexContent>
  </xs:complexType>
</xs:element>
<xs:element name='setNodeParametersAfterEditResponse' type='
    SetNodeParametersResponse'/>

<xs:element name='getCommonNodeParametersForEdit' type='
    NodeReferencesRequest'/>
<xs:element name='getCommonNodeParametersForEditResponse' type='
    ParameterFormResponse'/>

<xs:element name='setCommonNodeParametersAfterEdit'>
  <xs:complexType>
    <xs:complexContent>
      <xs:extension base='NodeReferencesRequest'>
        <xs:sequence>
          <xs:element ref='xd:x' minOccurs="1" maxOccurs="1"/>
        </xs:sequence>
      </xs:extension>
```

```
    </xs:complexContent>
  </xs:complexType>
</xs:element>
<xs:element name='setCommonNodeParametersAfterEditResponse' type='
    SetNodeParametersResponse'/>

<xs:element name='getAddableNodeTypes' type='NodeReferenceRequest'/>
<xs:element name='getAddableNodeTypesResponse'>
  <xs:complexType>
    <xs:sequence minOccurs='0' maxOccurs='unbounded'>
      <xs:element name='nodeType'>
        <xs:complexType>
          <xs:attribute name='type' type='xs:string' use='required'/
              >
          <xs:attribute name='name' type='xs:string' use='required'/
              >
        </xs:complexType>
      </xs:element>
    </xs:sequence>
  </xs:complexType>
</xs:element>

<xs:element name='getParametersForNewNode' type='
    NodeTypeReferenceRequest'/>
<xs:element name='getParametersForNewNodeResponse' type='
    ParameterFormResponse'/>

<xs:element name='createNewNode'>
  <xs:complexType>
    <xs:complexContent>
      <xs:extension base='NodeTypeReferenceRequest'>
        <xs:sequence>
          <xs:element ref='xd:x' minOccurs="1" maxOccurs="1"/>
        </xs:sequence>
      </xs:extension>
    </xs:complexContent>
  </xs:complexType>
</xs:element>
<xs:element name='createNewNodeResponse' type='
    SetNodeParametersResponse'/>

<xs:element name='destroyNode' type='NodeReferenceRequest'/>
<xs:element name='destroyNodeResponse'>
  <xs:complexType>
    <xs:attributeGroup ref='responseCode'/>
  </xs:complexType>
</xs:element>
```

```
<xs:element name='getAncestors' type='NodeReferenceParametersRequest
    '/>
<xs:element name='getAncestorsResponse' type='
    NodeInformationArrayResponse'/>

<xs:element name='getNodeCommands' type='NodeReferenceRequest'/>
<xs:element name='getNodeCommandsResponse' type='CommandArray'/>

<xs:element name='getCommandParameters' type='
    NodeCommandReferenceRequest'/>
<xs:element name='getCommandParametersResponse' type='
    ParameterFormResponse'/>

<xs:element name='executeNodeCommand'>
  <xs:complexType>
    <xs:complexContent>
      <xs:extension base='NodeCommandReferenceRequest'>
        <xs:sequence>
          <xs:element ref='xd:x' minOccurs="0" maxOccurs="1"/>
        </xs:sequence>
      </xs:extension>
    </xs:complexContent>
  </xs:complexType>
</xs:element>
<xs:element name='executeNodeCommandResponse' type='
    ExecuteNodeCommandResponse'/>

<xs:element name='executeNodeQuery'>
  <xs:complexType>
    <xs:complexContent>
      <xs:extension base='NodeCommandReferenceRequest'>
        <xs:sequence>
          <xs:element ref='xd:x' minOccurs="0" maxOccurs="1"/>
        </xs:sequence>
      </xs:extension>
    </xs:complexContent>
  </xs:complexType>
</xs:element>
<xs:element name='executeNodeQueryResponse'>
  <xs:complexType>
    <xs:complexContent>
      <xs:extension base='ExecuteNodeCommandResponse'>
        <xs:attribute name="queryId" type="xs:string" use="optional"
            />
      </xs:extension>
    </xs:complexContent>
  </xs:complexType>
</xs:element>
```

```
<xs:element name='getCommonNodeCommands' type='NodeReferencesRequest
    '/>
<xs:element name='getCommonNodeCommandsResponse' type='CommandArray'
    />

<xs:element name='getCommonCommandParameters'>
  <xs:complexType>
    <xs:complexContent>
      <xs:extension base='NodeReferencesRequest'>
        <xs:attribute name='command' type='xs:string' use='required'
            />
      </xs:extension>
    </xs:complexContent>
  </xs:complexType>
</xs:element>
<xs:element name='getCommonCommandParametersResponse' type='
    ParameterFormResponse'/>

<xs:element name='executeCommonNodeCommand'>
  <xs:complexType>
    <xs:complexContent>
      <xs:extension base='NodeReferencesRequest'>
        <xs:sequence>
          <xs:element ref='xd:x' minOccurs="0" maxOccurs="1"/>
        </xs:sequence>
        <xs:attribute name="queryId" type="xs:string" use="optional"
            />
      </xs:extension>
    </xs:complexContent>
  </xs:complexType>
</xs:element>
<xs:element name='executeCommonNodeCommandResponse' type='
    ExecuteCommonNodeCommandResponse'/>

<xs:element name='executeCommonNodeQuery'>
  <xs:complexType>
    <xs:complexContent>
      <xs:extension base='NodeReferencesRequest'>
        <xs:sequence>
          <xs:element ref='xd:x' minOccurs="0" maxOccurs="1"/>
        </xs:sequence>
      </xs:extension>
    </xs:complexContent>
  </xs:complexType>
</xs:element>
<xs:element name='executeCommonNodeQueryResponse'>
  <xs:complexType>
    <xs:complexContent>
      <xs:extension base='ExecuteCommonNodeCommandResponse'>
```

```
        <xs:attribute name="queryId" type="xs:string" use="optional"
            />
      </xs:extension>
    </xs:complexContent>
  </xs:complexType>
</xs:element>

<xs:element name='moveNodeUp' type='NodeReferenceRequest'/>
<xs:element name='moveNodeUpResponse' type='Response'/>

<xs:element name='moveNodeDown' type='NodeReferenceRequest'/>
<xs:element name='moveNodeDownResponse' type='Response'/>

<xs:element name='moveNodesUp' type='NodeReferencesRequest'/>
<xs:element name='moveNodesUpResponse' type='Response'/>

<xs:element name='moveNodesDown' type='NodeReferencesRequest'/>
<xs:element name='moveNodesDownResponse' type='Response'/>

<xs:element name='subscribe'>
  <xs:complexType>
    <xs:complexContent>
      <xs:extension base='SourceReferenceRequest'>
        <xs:attribute name='getEventsSince' type='xs:dateTime' use='
            optional'/>
        <xs:attributeGroup ref='parametersAndMessages'/>
        <xs:attributeGroup ref='eventTypes'/>
      </xs:extension>
    </xs:complexContent>
  </xs:complexType>
</xs:element>
<xs:element name='subscribeResponse' type='Response'/>

<xs:element name='unsubscribe'>
  <xs:complexType>
    <xs:complexContent>
      <xs:extension base='SourceReferenceRequest'>
        <xs:attributeGroup ref='eventTypes'/>
      </xs:extension>
    </xs:complexContent>
  </xs:complexType>
</xs:element>
<xs:element name='unsubscribeResponse' type='Response'/>

<xs:element name='nodeAdded'>
  <xs:complexType>
    <xs:complexContent>
      <xs:extension base='NodeSourceInformation'>
```

```
        <xs:attribute name='afterNodeId' type='xs:string' use='
            optional'/>
        <xs:attribute name='afterNodeCacheType' type='xs:string' use
            ='optional'/>
      </xs:extension>
    </xs:complexContent>
  </xs:complexType>
</xs:element>

<xs:element name='nodeUpdated'>
  <xs:complexType>
    <xs:complexContent>
      <xs:extension base='NodeSourceInformation'>
        <xs:attribute name='oldId' type='xs:string' use='optional'/>
      </xs:extension>
    </xs:complexContent>
  </xs:complexType>
</xs:element>

<xs:element name='nodeStatusChanged'>
  <xs:complexType>
    <xs:complexContent>
      <xs:extension base='NodeReference'>
        <xs:choice minOccurs='0' maxOccurs='unbounded'>
          <xs:element name='message' type='Message'/>
        </xs:choice>
        <xs:attribute name='state' type='NodeState' use='required'/>
      </xs:extension>
    </xs:complexContent>
  </xs:complexType>
</xs:element>

<xs:element name='nodeRemoved' type='NodeReference'/>
<xs:element name='nodeMovedUp' type='NodeReference'/>
<xs:element name='nodeMovedDown' type='NodeReference'/>

<xs:element name='getDatabases' type='TokenRequest'/>
<xs:element name='getDatabasesResponse'>
  <xs:complexType>
    <xs:sequence minOccurs='0' maxOccurs='unbounded'>
      <xs:element name='database'>
        <xs:complexType>
          <xs:attribute name='databaseId' type='xs:string' use='
              required'/>
          <xs:attribute name='name' type='xs:string' use='optional'/
              >
        </xs:complexType>
      </xs:element>
    </xs:sequence>
```

```
    <xs:attributeGroup ref='responseCode'/>
  </xs:complexType>
</xs:element>

<xs:element name='getDatabaseReadoutParameters' type='
   DatabaseReferenceRequest'/>
<xs:element name='getDatabaseReadoutParametersResponse' type='
   ParameterFormResponse'/>

<xs:element name='startDatabaseReadout'>
  <xs:complexType>
    <xs:complexContent>
      <xs:extension base='DatabaseReferenceRequest'>
        <xs:sequence>
          <xs:element name='node' type='NodeReference' minOccurs='1'
              maxOccurs='unbounded'/>
          <xs:element ref='xd:x' minOccurs="1" maxOccurs="1"/>
        </xs:sequence>
        <xs:attribute name="seqnr" type="xs:int" use="required"/>
      </xs:extension>
    </xs:complexContent>
  </xs:complexType>
</xs:element>
<xs:element name='startDatabaseReadoutResponse' type='
   ExecuteNodeCommandResponse'/>

<xs:element name='queryProgress'>
  <xs:complexType>
    <xs:choice minOccurs='1' maxOccurs='unbounded'>
      <xs:element name='queryStarted'>
        <xs:complexType/>
      </xs:element>
      <xs:element name='queryDone'>
        <xs:complexType/>
      </xs:element>
      <xs:element name='queryAborted'>
        <xs:complexType/>
      </xs:element>
      <xs:element name='newTable'>
        <xs:complexType>
          <xs:sequence minOccurs='0' maxOccurs='unbounded'>
            <xs:element name='column'>
              <xs:complexType>
                <xs:attribute name='columnId' type='xs:string' use='
                   required'/>
                <xs:attribute name='header' type='xs:string' use='
                   optional'/>
                <xs:attribute name='dataSourceId' type='xs:string'
                   use='optional'/>
```

```
                <xs:attribute name='cacheTypeName' type='xs:string'
                    use='optional'/>
                <xs:attribute name='fgColor' type='Color' use='
                    optional'/>
                <xs:attribute name='bgColor' type='Color' use='
                    optional'/>
                <xs:attribute name='alignment' type='Alignment' use=
                    'optional'/>
                <xs:attribute name='nrDecimals' type='
                    xs:nonNegativeInteger' use='optional'/>
              </xs:complexType>
            </xs:element>
          </xs:sequence>
          <xs:attribute name='tableId' type='xs:string' use='
              required'/>
          <xs:attribute name='tableName' type='xs:string' use='
              required'/>
        </xs:complexType>
      </xs:element>
      <xs:element name='newRecords'>
        <xs:complexType>
          <xs:sequence minOccurs='0' maxOccurs='unbounded'>
            <xs:element name='record'>
              <xs:complexType>
                <xs:choice minOccurs='0' maxOccurs='unbounded'>
                  <xs:element name='boolean' type='xs:boolean'/>
                  <xs:element name='color' type='Color'/>
                  <xs:element name='date' type='xs:date'/>
                  <xs:element name='dateTime' type='xs:dateTime'/>
                  <xs:element name='double' type='xs:double'/>
                  <xs:element name='duration' type='xs:duration'/>
                  <xs:element name='int' type='xs:int'/>
                  <xs:element name='long' type='xs:long'/>
                  <xs:element name='string' type='xs:string'/>
                  <xs:element name='time' type='xs:time'/>
                  <xs:element name='base64'>
                    <xs:complexType>
                      <xs:simpleContent>
                        <xs:extension base='xs:base64Binary'>
                          <xs:attribute name='contentType' type='
                              xs:string' use='required'/>
                        </xs:extension>
                      </xs:simpleContent>
                    </xs:complexType>
                  </xs:element>
                  <xs:element name='void'>
                    <xs:complexType/>
                  </xs:element>
                </xs:choice>
```

```
            </xs:complexType>
          </xs:element>
        </xs:sequence>
        <xs:attribute name='tableId' type='xs:string' use='
            required'/>
      </xs:complexType>
    </xs:element>
    <xs:element name='tableDone'>
      <xs:complexType>
        <xs:attribute name='tableId' type='xs:string' use='
            required'/>
      </xs:complexType>
    </xs:element>
    <xs:element name='newObject'>
      <xs:complexType>
        <xs:simpleContent>
          <xs:extension base='xs:base64Binary'>
            <xs:attribute name='contentType' type='xs:string' use=
                'required'/>
          </xs:extension>
        </xs:simpleContent>
      </xs:complexType>
    </xs:element>
    <xs:element name='queryMessage'>
      <xs:complexType>
        <xs:simpleContent>
          <xs:extension base='xs:string'>
            <xs:attribute name='type' type='EventType' use='
                optional' default='Information'/>
            <xs:attribute name='level' type='EventLevel' use='
                optional' default='Minor'/>
          </xs:extension>
        </xs:simpleContent>
      </xs:complexType>
    </xs:element>
    <xs:element name='title'>
      <xs:complexType>
        <xs:attribute name='name' type='xs:string' use='required'/
            >
      </xs:complexType>
    </xs:element>
    <xs:element name='status'>
      <xs:complexType>
        <xs:attribute name='message' type='xs:string' use='
            required'/>
      </xs:complexType>
    </xs:element>
    <xs:element name='beginSection'>
      <xs:complexType>
```

```
            <xs:attribute name='header' type='xs:string' use='required
                '/>
          </xs:complexType>
        </xs:element>
        <xs:element name='endSection'>
          <xs:complexType/>
        </xs:element>
      </xs:choice>
      <xs:attributeGroup ref='nodeReference'/>
      <xs:attribute name='queryId' type='xs:string' use='required'/>
    </xs:complexType>
  </xs:element>

  <xs:element name='abortNodeQuery'>
    <xs:complexType>
      <xs:complexContent>
        <xs:extension base='NodeReferenceRequest'>
          <xs:attribute name='queryId' type='xs:string' use='required'
              />
        </xs:extension>
      </xs:complexContent>
    </xs:complexType>
  </xs:element>
  <xs:element name='abortNodeQueryResponse'>
    <xs:complexType>
      <xs:attributeGroup ref='responseCode'/>
    </xs:complexType>
  </xs:element>

  <xs:element name='abortCommonNodeQuery'>
    <xs:complexType>
      <xs:complexContent>
        <xs:extension base='NodeReferencesRequest'>
          <xs:attribute name='queryId' type='xs:string' use='required'
              />
        </xs:extension>
      </xs:complexContent>
    </xs:complexType>
  </xs:element>
  <xs:element name='abortCommonNodeQueryResponse'>
    <xs:complexType>
      <xs:attributeGroup ref='responseCode'/>
    </xs:complexType>
  </xs:element>

  <xs:simpleType name='ResponseCode'>
    <xs:restriction base='xs:string'>
      <xs:enumeration value='OK'/>
      <xs:enumeration value='NotFound'/>
```

```
      <xs:enumeration value='InsufficientPrivileges'/>
      <xs:enumeration value='Locked'/>
      <xs:enumeration value='NotImplemented'/>
      <xs:enumeration value='FormError'/>
      <xs:enumeration value='OtherError'/>
    </xs:restriction>
  </xs:simpleType>

  <xs:simpleType name='Color'>
    <xs:restriction base='xs:string'>
      <xs:pattern value='^[0-9a-fA-F]{6}$'/>
    </xs:restriction>
  </xs:simpleType>

  <xs:simpleType name='NodeState'>
    <xs:restriction base='xs:string'>
      <xs:enumeration value='None'/>
      <xs:enumeration value='Information'/>
      <xs:enumeration value='WarningSigned'/>
      <xs:enumeration value='WarningUnsigned'/>
      <xs:enumeration value='ErrorSigned'/>
      <xs:enumeration value='ErrorUnsigned'/>
    </xs:restriction>
  </xs:simpleType>

  <xs:simpleType name='MessageType'>
    <xs:restriction base='xs:string'>
      <xs:enumeration value='Error'/>
      <xs:enumeration value='Warning'/>
      <xs:enumeration value='Information'/>
    </xs:restriction>
  </xs:simpleType>

  <xs:simpleType name='CommandType'>
    <xs:restriction base='xs:string'>
      <xs:enumeration value='Simple'/>
      <xs:enumeration value='Parameterized'/>
      <xs:enumeration value='Query'/>
    </xs:restriction>
  </xs:simpleType>

  <xs:simpleType name='Alignment'>
    <xs:restriction base='xs:string'>
      <xs:enumeration value='Left'/>
      <xs:enumeration value='Center'/>
      <xs:enumeration value='Right'/>
    </xs:restriction>
  </xs:simpleType>
```

```
<xs:simpleType name='EventType'>
  <xs:restriction base='xs:string'>
    <xs:enumeration value='Information'/>
    <xs:enumeration value='Warning'/>
    <xs:enumeration value='Error'/>
    <xs:enumeration value='Exception'/>
  </xs:restriction>
</xs:simpleType>

<xs:simpleType name='EventLevel'>
  <xs:restriction base='xs:string'>
    <xs:enumeration value='Minor'/>
    <xs:enumeration value='Medium'/>
    <xs:enumeration value='Major'/>
  </xs:restriction>
</xs:simpleType>

<xs:attributeGroup name='tokens'>
  <xs:attribute name='deviceToken' type='xs:string' use='optional'/>
  <xs:attribute name='serviceToken' type='xs:string' use='optional'/
     >
  <xs:attribute name='userToken' type='xs:string' use='optional'/>
</xs:attributeGroup>

<xs:attributeGroup name='sourceReference'>
  <xs:attribute name='sourceId' type='xs:string' use='required'/>
</xs:attributeGroup>

<xs:attributeGroup name='nodeReference'>
  <xs:attributeGroup ref='sourceReference'/>
  <xs:attribute name='nodeId' type='xs:string' use='required'/>
  <xs:attribute name='cacheType' type='xs:string' use='optional'/>
</xs:attributeGroup>

<xs:attributeGroup name='parametersAndMessages'>
  <xs:attribute name='parameters' type='xs:boolean' use='optional'
     default='false'/>
  <xs:attribute name='messages' type='xs:boolean' use='optional'
     default='false'/>
</xs:attributeGroup>

<xs:attributeGroup name='indexReference'>
  <xs:attributeGroup ref='sourceReference'/>
  <xs:attribute name='index' type='xs:string' use='required'/>
  <xs:attribute name='indexValue' type='xs:string' use='required'/>
</xs:attributeGroup>

<xs:attributeGroup name='responseCode'>
  <xs:attribute name='result' type='ResponseCode' use='required'/>
```

```
</xs:attributeGroup>

<xs:attributeGroup name='eventTypes'>
  <xs:attribute name='nodeAdded' type='xs:boolean' use='optional'
     default='true'/>
  <xs:attribute name='nodeUpdated' type='xs:boolean' use='optional'
     default='true'/>
  <xs:attribute name='nodeStatusChanged' type='xs:boolean' use='
     optional' default='true'/>
  <xs:attribute name='nodeRemoved' type='xs:boolean' use='optional'
     default='true'/>
  <xs:attribute name='nodeMovedUp' type='xs:boolean' use='optional'
     default='true'/>
  <xs:attribute name='nodeMovedDown' type='xs:boolean' use='optional
     ' default='true'/>
</xs:attributeGroup>

<xs:attributeGroup name='databaseReference'>
  <xs:attribute name='databaseId' type='xs:string' use='required'/>
</xs:attributeGroup>

<xs:attributeGroup name='readoutReference'>
  <xs:attribute name='seqnr' type='xs:int' use='required'/>
</xs:attributeGroup>

<xs:complexType name='Response'>
  <xs:attributeGroup ref='responseCode'/>
</xs:complexType>

<xs:complexType name='TokenRequest'>
  <xs:attributeGroup ref='tokens'/>
</xs:complexType>

<xs:complexType name='SourceReferenceRequest'>
  <xs:complexContent>
    <xs:extension base='TokenRequest'>
      <xs:attributeGroup ref='sourceReference'/>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>

<xs:complexType name='DataSource'>
  <xs:attribute name='sourceId' type='xs:string' use='required'/>
  <xs:attribute name='name' type='xs:string' use='required'/>
  <xs:attribute name='hasChildren' type='xs:boolean' use='optional'
     default='false'/>
  <xs:attribute name='lastChanged' type='xs:dateTime' use='optional'
     />
</xs:complexType>
```

```
<xs:complexType name='DataSourceArrayResponse'>
  <xs:sequence minOccurs='0' maxOccurs='unbounded'>
    <xs:element name='dataSource' type='DataSource'/>
  </xs:sequence>
  <xs:attributeGroup ref='responseCode'/>
</xs:complexType>

<xs:complexType name='NodeReferenceRequest'>
  <xs:complexContent>
    <xs:extension base='TokenRequest'>
      <xs:attributeGroup ref='nodeReference'/>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>

<xs:complexType name='NodeReference'>
  <xs:attributeGroup ref='nodeReference'/>
</xs:complexType>

<xs:complexType name='NodeReferencesRequest'>
  <xs:complexContent>
    <xs:extension base='TokenRequest'>
      <xs:sequence minOccurs='0' maxOccurs='unbounded'>
        <xs:element name='node' type='NodeReference'/>
      </xs:sequence>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>

<xs:complexType name='NodeReferenceParametersRequest'>
  <xs:complexContent>
    <xs:extension base='NodeReferenceRequest'>
      <xs:attributeGroup ref='parametersAndMessages'/>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>

<xs:complexType name='SourceReferenceParametersRequest'>
  <xs:complexContent>
    <xs:extension base='SourceReferenceRequest'>
      <xs:attributeGroup ref='parametersAndMessages'/>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>

<xs:complexType name='GetAllNodesRequest'>
  <xs:complexContent>
    <xs:extension base='SourceReferenceParametersRequest'>
```

```
      <xs:sequence minOccurs='0' maxOccurs='unbounded'>
        <xs:element name='onlyIfDerivedFrom' type='xs:string'/>
      </xs:sequence>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>

<xs:complexType name='GetChildNodesRequest'>
  <xs:complexContent>
    <xs:extension base='NodeReferenceParametersRequest'>
      <xs:sequence minOccurs='0' maxOccurs='unbounded'>
        <xs:element name='sortOrder' type='SortOrder'/>
      </xs:sequence>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>

<xs:complexType name='SortOrder'>
  <xs:attribute name='parameterName' type='xs:string' use='required'
     />
  <xs:attribute name='ascending' type='xs:boolean' use='optional'
     default='true'/>
</xs:complexType>

<xs:complexType name='NodeInformation'>
  <xs:choice minOccurs='0' maxOccurs='unbounded'>
    <xs:element name='boolean' type='BooleanParameter'/>
    <xs:element name='color' type='ColorParameter'/>
    <xs:element name='dateTime' type='DateTimeParameter'/>
    <xs:element name='double' type='DoubleParameter'/>
    <xs:element name='duration' type='DurationParameter'/>
    <xs:element name='int' type='IntParameter'/>
    <xs:element name='long' type='LongParameter'/>
    <xs:element name='string' type='StringParameter'/>
    <xs:element name='time' type='TimeParameter'/>
    <xs:element name='message' type='Message'/>
  </xs:choice>
  <xs:attribute name='nodeId' type='xs:string' use='required'/>
  <xs:attribute name='displayName' type='xs:string' use='optional'/>
  <xs:attribute name='nodeType' type='xs:string' use='optional'/>
  <xs:attribute name='localId' type='xs:string' use='optional'/>
  <xs:attribute name='logId' type='xs:string' use='optional'/>
  <xs:attribute name='cacheType' type='xs:string' use='optional'/>
  <xs:attribute name='state' type='NodeState' use='required'/>
  <xs:attribute name='hasChildren' type='xs:boolean' use='required'/
     >
  <xs:attribute name='childrenOrdered' type='xs:boolean' use='
     optional' default='false'/>
```

```
    <xs:attribute name='isReadable' type='xs:boolean' use='optional'
        default='false'/>
    <xs:attribute name='isControllable' type='xs:boolean' use='
        optional' default='false'/>
    <xs:attribute name='hasCommands' type='xs:boolean' use='optional'
        default='false'/>
    <xs:attribute name='parentId' type='xs:string' use='optional'/>
    <xs:attribute name='parentCacheType' type='xs:string' use='
        optional'/>
    <xs:attribute name='lastChanged' type='xs:dateTime' use='optional'
        />
  </xs:complexType>

  <xs:complexType name='NodeSourceInformation'>
    <xs:complexContent>
      <xs:extension base='NodeInformation'>
        <xs:attributeGroup ref='sourceReference'/>
      </xs:extension>
    </xs:complexContent>
  </xs:complexType>

  <xs:complexType name='Parameter' abstract='true'>
    <xs:attribute name='id' type='xs:string' use='required'/>
    <xs:attribute name='name' type='xs:string' use='required'/>
  </xs:complexType>

  <xs:complexType name='BooleanParameter'>
    <xs:complexContent>
      <xs:extension base='Parameter'>
        <xs:attribute name='value' type='xs:boolean' use='required'/>
      </xs:extension>
    </xs:complexContent>
  </xs:complexType>

  <xs:complexType name='ColorParameter'>
    <xs:complexContent>
      <xs:extension base='Parameter'>
        <xs:attribute name='value' type='Color' use='required'/>
      </xs:extension>
    </xs:complexContent>
  </xs:complexType>

  <xs:complexType name='DateTimeParameter'>
    <xs:complexContent>
      <xs:extension base='Parameter'>
        <xs:attribute name='value' type='xs:dateTime' use='required'/>
      </xs:extension>
    </xs:complexContent>
  </xs:complexType>
```

```
<xs:complexType name='DoubleParameter'>
  <xs:complexContent>
    <xs:extension base='Parameter'>
      <xs:attribute name='value' type='xs:double' use='required'/>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>

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

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

<xs:complexType name='LongParameter'>
  <xs:complexContent>
    <xs:extension base='Parameter'>
      <xs:attribute name='value' type='xs:long' use='required'/>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>

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

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

```
<xs:complexType name='Message'>
  <xs:simpleContent>
    <xs:extension base='xs:string'>
      <xs:attribute name='timestamp' type='xs:dateTime' use='
          required'/>
      <xs:attribute name='type' type='MessageType' use='required'/>
      <xs:attribute name='eventId' type='xs:string' use='optional'/>
    </xs:extension>
  </xs:simpleContent>
</xs:complexType>

<xs:complexType name='NodeInformationArrayResponse'>
  <xs:sequence minOccurs='0' maxOccurs='unbounded'>
    <xs:element name='node' type='NodeInformation'/>
  </xs:sequence>
  <xs:attributeGroup ref='responseCode'/>
</xs:complexType>

<xs:complexType name='StringArray'>
  <xs:sequence minOccurs='0' maxOccurs='unbounded'>
    <xs:element name='value' type='xs:string'/>
  </xs:sequence>
</xs:complexType>

<xs:complexType name='StringArrayResponse'>
  <xs:complexContent>
    <xs:extension base='StringArray'>
      <xs:attributeGroup ref='responseCode'/>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>

<xs:complexType name='BooleanArrayResponse'>
  <xs:sequence minOccurs='0' maxOccurs='unbounded'>
    <xs:element name='value' type='xs:boolean'/>
  </xs:sequence>
  <xs:attributeGroup ref='responseCode'/>
</xs:complexType>

<xs:complexType name='IndexReferenceRequest'>
  <xs:complexContent>
    <xs:extension base='TokenRequest'>
      <xs:attributeGroup ref='indexReference'/>
      <xs:attributeGroup ref='parametersAndMessages'/>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>

<xs:complexType name='IndexReference'>
```

```
    <xs:attributeGroup ref='indexReference'/>
  </xs:complexType>

  <xs:complexType name='IndexReferencesRequest'>
    <xs:complexContent>
      <xs:extension base='TokenRequest'>
        <xs:sequence minOccurs='0' maxOccurs='unbounded'>
          <xs:element name='indexRef' type='IndexReference'/>
        </xs:sequence>
        <xs:attributeGroup ref='parametersAndMessages'/>
      </xs:extension>
    </xs:complexContent>
  </xs:complexType>

  <xs:complexType name='SetNodeParametersResponse'>
    <xs:choice minOccurs='1' maxOccurs='1'>
      <xs:element name='error' type='ParameterError' minOccurs='1'
          maxOccurs='unbounded'/>
      <xs:element name='node' type='NodeInformation' minOccurs='1'
          maxOccurs='unbounded'/>
    </xs:choice>
    <xs:attributeGroup ref='responseCode'/>
  </xs:complexType>

  <xs:complexType name='ParameterFormResponse'>
    <xs:sequence>
      <xs:element ref='xd:x' minOccurs="1" maxOccurs="1"/>
    </xs:sequence>
    <xs:attributeGroup ref="responseCode"/>
  </xs:complexType>

  <xs:complexType name='NodeTypeReferenceRequest'>
    <xs:complexContent>
      <xs:extension base='NodeReferenceRequest'>
        <xs:attribute name='type' type='xs:string' use='required'/>
      </xs:extension>
    </xs:complexContent>
  </xs:complexType>

  <xs:complexType name='Command'>
    <xs:attribute name='command' type='xs:string' use='required'/>
    <xs:attribute name='name' type='xs:string' use='required'/>
    <xs:attribute name='type' type='CommandType' use='required'/>
    <xs:attribute name='sortCategory' type='xs:string' use='optional'/
        >
    <xs:attribute name='sortKey' type='xs:string' use='optional'/>
    <xs:attribute name='confirmationString' type='xs:string' use='
        optional'/>
```

```
    <xs:attribute name='failureString' type='xs:string' use='optional'
        />
    <xs:attribute name='successString' type='xs:string' use='optional'
        />
  </xs:complexType>

  <xs:complexType name='CommandArray'>
    <xs:sequence minOccurs='0' maxOccurs='unbounded'>
      <xs:element name='command' type='Command'/>
    </xs:sequence>
  </xs:complexType>

  <xs:complexType name='NodeCommandReferenceRequest'>
    <xs:complexContent>
      <xs:extension base='NodeReferenceRequest'>
        <xs:attribute name='command' type='xs:string' use='required'/>
      </xs:extension>
    </xs:complexContent>
  </xs:complexType>

  <xs:complexType name='ExecuteNodeCommandResponse'>
    <xs:choice minOccurs='0' maxOccurs='unbounded'>
      <xs:element name='error' type='ParameterError'/>
    </xs:choice>
    <xs:attributeGroup ref='responseCode'/>
  </xs:complexType>

  <xs:complexType name='ExecuteCommonNodeCommandResponse'>
    <xs:complexContent>
      <xs:extension base='ExecuteNodeCommandResponse'>
        <xs:choice minOccurs='0' maxOccurs='unbounded'>
          <xs:element name='result'>
            <xs:complexType>
              <xs:simpleContent>
                <xs:extension base='xs:boolean'>
                  <xs:attribute name='error' type='xs:string' use='
                      optional'/>
                </xs:extension>
              </xs:simpleContent>
            </xs:complexType>
          </xs:element>
        </xs:choice>
      </xs:extension>
    </xs:complexContent>
  </xs:complexType>

  <xs:complexType name='ParameterError'>
    <xs:simpleContent>
      <xs:extension base='xs:string'>
```

```
        <xs:attribute name='var' type='xs:string' use='required'/>
      </xs:extension>
    </xs:simpleContent>
  </xs:complexType>

  <xs:complexType name='ReadoutReference'>
    <xs:attributeGroup ref='databaseReference'/>
    <xs:attributeGroup ref='readoutReference'/>
  </xs:complexType>

  <xs:complexType name='DatabaseReferenceRequest'>
    <xs:complexContent>
      <xs:extension base='TokenRequest'>
        <xs:attributeGroup ref='databaseReference'/>
      </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 Joachim Lindborg, Karin Forsell, Tina Beckman and Klaudiusz Staniek for all valuable feedback.