XEP-0324: Internet of Things - Provisioning

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

2017-05-20
Version 0.5

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

Note: This specification has been retracted by the author; new implementations are not recommended. This specification describes an architecture for efficient provisioning of services, access rights and user privileges in for the Internet of Things, where communication between Things is done using the XMPP protocol.

Legal

Copyright

Permissions

Warranty

Liability

Conformance

Contents

1 Introduction

This specification describes an architecture for efficient provisioning of services, access rights and user privileges in for the Internet of Things, where communication between Things is done using the XMPP protocol.
Note has to be taken, that this XEP, and other Internet of Things-related XEP's, are designed for implementation in small devices, many of which have very limited amount of memory (both RAM and ROM) or resources (processing power). Therefore, simplicity is of utmost importance. Furthermore, Internet of Things networks can become huge, easily containing millions or billions of devices in peer-to-peer networks.
An added complexity in the provisioning case is that Things (small sensors for example) often have very limited user interface options. Therefore, this document explains how provisioning can be done efficiently using a trusted third party with more power and options when it comes to user interface design and storage.
This document defines the following important operations to allow for efficient provisioning of services in the Internet of Things, based on XMPP:

- What Things knows what Things
- What Things can read data from what Things, and what data.
- What Things can control what Things, and what parts.
- Control of Users in the network.
- Control of Services in the network.
- Control generic boolean User Privileges in the network.

This XEP relies on Internet of Things - Sensor Data (XEP-0323) [1] and Internet of Things - Control (XEP-0325) [2] for sensor data readout and control interfaces. It relies on Internet of Things - Concentrators (XEP-0326) [3] for bridging protocols and interfaing entities with multiple devices behind them. It also ties into Internet of Things - Discovery (XEP-0347) [4] for automatic discovery of provisioning servers by things.
Internet of Things contain many different architectures and use cases. For this reason, the IoT standards have been divided into multiple XEPs according to the following table:

[1] XEP-0323: Internet of Things - Sensor Data <https://xmpp.org/extensions/xep-0323.html>.
[2] XEP-0325: Internet of Things - Control <https://xmpp.org/extensions/xep-0325.html>.
[3] XEP-0326: Internet of Things - Concentrators <https://xmpp.org/extensions/xep-0326.html>.
[4] XEP-0347: Internet of Things - Discovery <https://xmpp.org/extensions/xep-0347.html>.

XEP	Description
xep-0000-IoT-BatteryPoweredSensors	Defines how to handle the peculiars related to battery powered devices, and other devices intermittently available on the network.
xep-0000-IoT-Events	Defines how Things send events, how event subscription, hysteresis levels, etc., are configured.
xep-0000-IoT-Interoperability	Defines guidelines for how to achieve interoperability in Internet of Things, 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 Internet of Things.
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 an Internet of Things specific XEP, this XEP should be considered in all Internet of Things 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 Internet of Things XEPs.
XEP-0324	This specification. Defines how provisioning, the management of access privileges, etc., can be efficiently and easily implemented.
XEP-0325	Defines how to control actuators and other devices in Internet of Things.
XEP-0326	Defines how to handle architectures containing concentrators or servers handling multiple Things.
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	Description
XEP-0336	Defines extensions for how dynamic forms can be created, based on Data Forms (XEP-0004) XEP-0004: Data Forms <https://xmpp.org/extensions/xep-0004.html>., Data Forms Validation (XEP-0122) XEP-0122: Data Forms Validation <https://xmpp.org/extensions/xep-0122.html>., Publishing Stream Initiation Requests (XEP-0137) XEP-0137: Publishing Stream Initiation Requests <https://xmpp.org/extensions/xep-0137.html>. and Data Forms Layout (XEP-0141) XEP-0141: Data Forms Layout <https://xmpp.org/extensions/xep-0141.html>..
XEP-0347	Defines the peculiars of sensor discovery in sensor networks. Apart from discovering sensors by JID, it also defines how to discover sensors based on location, etc.

2 Glossary

The following table lists common terms and corresponding descriptions.

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

Authority Used synonymously with Provisioning Server.

Computed Value A value that is computed instead of measured.

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

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

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

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

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

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

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

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

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

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

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

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

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

Provisioning Server An application that can configure a network and provide services to users or Things. In Internet of Things, a Provisioning Server knows who knows whom, what privileges users have, who can read what data and who can control what devices and what parts of these devices.

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.

Thing Internet of Things basically consists of Things connected to the Internet. Things can be any device, sensor, actuator etc., that can have an Internet connection.

Thing Registry A registry where Things can register for simple and secure discovery by the owner of the Thing. The registry can also be used as a database for meta information about Things in the network.

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

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

Value A field value.

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

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

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

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

3 Use Cases

The most basic use case in sensor networks is to read out sensor data from a sensor. However, since protecting end-user integrity and system security is vital, access rights and user privileges have to be imposed on the network.
To store access rights in all sensors might be very impractical. Not only does it consume memory, it's difficult to maintain track of the current system status, make sure all devices have the latest configuration, distribute changes to the configuration, etc.
Furthermore, most sensors and small devices have very limited possibility to provide a rich user interface. Perhaps all it can do is to provide a small LED and a button, useful perhaps for installing the sensor in the network, but not much more.
As an added complexity, the sensor network operator might not even have access to the XMPP Servers used, and provisioning needs to lie outside of the XMPP Server domains.
To solve this problem in an efficient manner, an architecture using distributed trusted third parties is proposed. Such third parties would:

- Provide a rich user interface and configurable options to end user or back end systems.

- Control friendships (who can communicate with whom).
- Control content available for different friends (what can be read by whom).
- Control operations accessible by different friends (what can be controlled/configured by whom).
- Provide additional interoperability services to nodes in the network (for instance, unit conversion).

3.1 Delegating trust

3.1.1 Delegating original trust to a Provisioning Server

A provisioning server can be accessed either through a JID published by the provisioning server, or through a subdomain address, if hosted as a server component. This section will show how to delegate original trust to a Provisioning Server, in the case the server uses a JID to communicate with things.
Trust is delegated to a provisioning server by a device, simply by befriending the provisioning server and asking it questions and complying with its answers. As an illustrative example, following is a short description of how such a trust relationship can be created in a scenario where the sensor only has a single LED and a single button.

- Somebody is installing the sensor, giving it a connection to an XMPP server and a JID, reachable from the provisioning server.
- The provisioning server is told to create a friendship request to the new sensor.
- The sensor flashes its LED for a given time (for example: 30 seconds).
- Viewing the LED, the person installing the sensor presses the button.
- Receiving the button press within the given time period, accepts the friendship request. Optionally, the device can give user feedback using the LED.
- The device performs a service discovery of the new friend, having been a manually added friend.
- If the new friend supports this provisioning extension, further responsibilities are delegated to this device.
- As the last step the device asks the provisioning server for a token. This device token is later used in calls to other devices and can be used to check access rights.

The following diagram shows the general case:

The successful case can also be illustrated in a sequence diagram, as follows:

Note: In many cases, an address to a provisioning server might be preprogrammed during production of the device. In these cases, parts of the above procedure may not be necessary. All the client needs to do, if the provisioning server is not available in the roster of the device, is to send a subscription request to the provisioning server, to alert the server of the existence of the device, and possibly request a device token.
Note 2: A certificate token has an undefined lifetime. It can be reused across sessions.
The following use cases will assume such a trust relationship has been created between the corresponding device and the provisioning server.

3.1.2 Provisioning Server as a server component

A provisioning server can also be hosted as a server component, and in these cases be addressed by using the component address, or sub-domain address of the component. In this case, the client searches through the components hosted by the server to see if one of them is a Provisioning Server. There are no friendship requests and presence subscriptions necessary, when communicating with a Provisioning Server hosted as a server component.
To search for a Provisioning Server hosted as a component on an XMPP Server, you first request a list of available components, as follows:

Listing 1: Checking if server supports components

```
<iq from='device@example.org/device' to='example.org' type='get' id='1
    '>
  <query xmlns="http://jabber.org/protocol/disco#info"/>
</iq>

<iq type="result" id="1" from="example.org" to="device@example.org/
    device">
  <query xmlns="http://jabber.org/protocol/disco#info">
    ...
    <feature var="http://jabber.org/protocol/disco#items"/>
    ...
  </query>
</iq>
```

If components (items) are supported, a request for available components is made:

Listing 2: Requesting list of server components

```
<iq from='device@example.org/device' to='example.org' type='get' id='2
    '>
```

```
  <query xmlns="http://jabber.org/protocol/disco#items"/>
</iq>

<iq type="result" id="2" from="example.org" to="995
    fab3dd759452ca9c370647323af0c@example.org/ebe2348e">
  <query xmlns="http://jabber.org/protocol/disco#items">
    ...
    <item jid="provisioning.example.org" name="Provisioning"/>
    ...
  </query>
</iq>
```

The client then loops through all components (items) and checks what features they support, until a Provisioning Server is found:

Listing 3: Service discovery information request made to each component

```
<iq type='get'
  from='device@example.org/device'
  to='provisioning.example.org'
  id='3'>
  <query xmlns='http://jabber.org/protocol/disco#info'/>
</iq>

<iq type='result'
  from='provisioning.example.org'
  to='device@example.org/device'
  id='3'>
  <query xmlns='http://jabber.org/protocol/disco#info'>
    ...
    <feature var='urn:xmpp:iot:provisioning'/>
    ...
  </query>
</iq>
```

3.1.3 Tokens and X.509 Certificates

The provisioning server contains a set of rules defining what operation can take place and by whom, by participants in the network. Rules can be applied based on JIDs used, content affected, and also through device, service and user identities based on X.509 Certificates. In order for a service (for instance) to identify itself in the network, it uses an X.509 certificate. It sends the public part of this certificate to the provisioning server, and receives a token back in the form of a simple string. This token can then be used in requests and propagated through the network.
To validate that the sender is allowed to use the certificate using its token, it encrypts a challenge using the public part of the certificate and sends it to the sender of the token, who in turn decrypts it using the private part of the certificate and returns it to the server.

The provisioning server can also use the public part of the certificate to perform validation checks on the certificate itself. If the certificate becomes invalid, the provisioning server can invalidate any corresponding rules in the network. If the sender of a token cannot respond to a token challenge, the provisioning server can also refuse to allow the operation.
In case of multiple units being part of an operation, a token can be propagated in the network. For example, a service can read data from U1, who reads data from U2. The service provides a token to U1, who propagates this token in the request to U2. When U2 asks the provisioning server if the operation should be allowed or not, the server knows what entity originated the request. If the provisioning server wants to challenge U2, concerning the token, U2 propagates the challenge to U1, who propagates it to the service, who can resolve the challenge, returns the response back to U1 who returns the response to U2 who in turn returns it to the provisioning server.

The following example shows how a device or service can request a token from the provisioning server, by providing the base-64 encoded public part of an X.509 certificate. This step is optional, but can be used as a method to identify the device (or service), apart from the JID it is using. This might be useful if you want to assign a particular device or service privileges in the provisioning server, regardless of the JID it uses to perform the action.

Listing 4: Requesting a token

```
<iq type='get'
  from='device@example.org/device'
  to='provisioning.example.org'
  id='4'>
  <getToken xmlns='urn:xmpp:iot:provisioning'>BASE-64 ENCODED PUBLIC X
      .509 CERTIFICATE</getToken>
</iq>

<iq type='result'
  from='provisioning.example.org'
  to='device@example.org/device'
  id='4'>
  <getTokenChallenge xmlns='urn:xmpp:iot:provisioning' seqnr='1'>BASE
      -64 ENCODED CHALLENGE</getTokenChallenge>
</iq>

<iq type='get'
  from='device@example.org/device'
  to='provisioning.example.org'
  id='5'>
  <getTokenChallengeResponse xmlns='urn:xmpp:iot:provisioning' seqnr='
      1'>BASE-64 ENCODED RESPONSE</getTokenChallengeResponse>
</iq>
```

```
<iq type='result'
  from='provisioning.example.org'
  to='device@example.org/device'
  id='5'>
  <getTokenResponse xmlns='urn:xmpp:iot:provisioning' token='TOKEN'/>
</iq>
```

The **getToken** element contains the base-64 encoded public version of the certificate that is used to identify the device or service. The server responds with a challenge in a **getTokenChallenge** response. This challenge is also a base-64 encoded binary block of data, which corresponds to a random sequence of bytes that is then encrypted using the public certificate. Now, the device, or service, decrypts this challenge using the private part of the certificate, and returns the base-64 encoded decrypted version of the challenge back to the provisioning server using the **getTokenChallengeResponse** element. The provisioning server checks the response to the original random sequence of bytes. If equal, the provisioning server responds with a **getTokenResponse** result, containing the token (a string this time) that can be used when reference to the identity defined by the certificate has to be made. The provisioning server must not return tokens that contain white space characters.
If the response to the challenge is wrong, the server returns a **bad-request** error result, as is shown below.

Listing 5: Challenge reponse incorrect

```
<iq type='result'
  from='provisioning.example.org'
  to='device@example.org/device'
  id='5'>
  <error type='modify'>
    <bad-request xmlns='urn:ietf:params:xml:ns:xmpp-stanzas'/>
  </error>
</iq>
```

If the sequence number identifying the challenge is not found on the server, the server returns a **item-not-found** error result, as is shown below.

Listing 6: Challenge sequence number not found

```
<iq type='result'
  from='provisioning.example.org'
  to='device@example.org/device'
  id='5'>
  <error type='cancel'>
    <item-not-found xmlns='urn:ietf:params:xml:ns:xmpp-stanzas'/>
  </error>
</iq>
```

The server must retain the challenge in memory for at least one minute before assuming the challenge will go unresponded.
For reasons the provisioning server determines, it can challenge the use of a token in any of the requests made to it. This is done by sending a iq get stanza with a **tokenChallenge** to the party sending the token. This element contains both the token being challenged, and a binary challenge. This challenge is made up of a random block of data that is encrypted using the public certificate referred to by the token.
The receiver of the challenge, if it has access to the private certificate referenced, decrypts the challenge, and returns the decrypted binary block of data to the caller (i.e. the Provisioning Server in this case). If the decrypted block of data corresponds to the original random block of data encrypted, the sender of the token is considered to be allowed to use the token.
If the received of the challenge does not have access to the private certificate referenced, but used the token in a propagated request made to it, it can propagate the request to the original sender of the token. When the response is returned, it returns the response in turn to the sender of the challenge.
A challenge/response sequence can look as follows:

Listing 7: Requesting a token

```
<iq type='get'
  from='provisioning.example.org'
  to='device@example.org/device'
  id='6'>
  <tokenChallenge xmlns='urn:xmpp:iot:provisioning' token='TOKEN'>BASE
      -64 encoded challenge</tokenChallenge>
</iq>

<iq type='result'
  from='device@example.org/device'
  to='provisioning.example.org'
  id='6'>
  <tokenChallengeResponse xmlns='urn:xmpp:iot:provisioning'>BASE-64
      encoded response</tokenChallengeResponse>
</iq>
```

Note: It is important that a unit only responds to a **tokenChallenge** request from a JID to which the corresponding token has been sent. If a token challenge is received from a JID to which the token has not been sent the last minute, the following error message must be returned:

Listing 8: Invalid token challenge

```
<iq type='error'
  from='device@example.org/device'
  to='provisioning.example.org'
  id='6'>
  <error type='cancel'>
```

```
    <forbidden xmlns='urn:ietf:params:xml:ns:xmpp-stanzas'/>
  </error>
</iq>
```

3.1.4 Delegating Secondary Trust

The **isFriendResponse** element returned by the provisioning server contains an attribute **secondaryTrustAllowed** that is by default set to false. If the provisioning server has no problem with allowing multiple trust to be delegated by devices in the network, it can choose to set this attribute to true in the response. If true, the device knows it has the right to add its own friends, or to add secondary trust relationships.
The following diagram continues with the example given above, of how a sensor with a limited user interface, can allow to manually add new friends, including new trust relationships using a single LED and a button.

3.1.5 Multiple tokens

When multiple trust is used, the entity (client, user, service, etc.) has one token from each provisioning server. However, when sending a token to a third party, the sender does not know what provisioning server(s) the third party uses to check access rights and user privileges. Therefore, the client must send all tokens, separated by a space.
When a provisioning server receives a request containing multiple tokens, the most forgiving response must be returned.

Listing 9: Readout request using multiple tokens

```
<iq type='get'
  from='master@example.org/amr'
  to='device@example.org'
  id='7'>
  <req xmlns='urn:xmpp:iot:sensordata' momentary='true' serviceToken='
     SERVICETOKEN1␣SERVICETOKEN2' userToken='USERTOKEN1' seqnr='4'/>
</iq>

<iq type='get'
  from='device@example.org/device'
  to='provisioning.example.org'
  id='8'>
  <canRead xmlns='urn:xmpp:iot:provisioning' jid='master@example.org'
     serviceToken='SERVICETOKEN1␣SERVICETOKEN2' userToken='USERTOKEN1
     ' momentary='true'/>
</iq>
```

```
<iq type='result'
  from='provisioning.example.org'
  to='device@example.org/device'
  id='8'>
  <canReadResponse xmlns='urn:xmpp:iot:provisioning' jid='
      master@example.org' momentary='true' result='true'/>
</iq>

<iq type='result'
  from='device@example.org'
  to='master@example.org/amr'
  id='7'>
  <accepted xmlns='urn:xmpp:iot:sensordata' seqnr='4'/>
</iq>
```

Note: When a provisioning server wants to challenge multiple tokens, separate token challenges are sent, one for each token being challenged.

3.2 Friendships

3.2.1 Friendship request accepted

The following diagram displays how a friendship request from an external party can be handled, delegating the responsibility to a trusted third party:

The communication between the XMPP Device and the Provisioning Server could be as follows:

Listing 10: Friendship request accepted

```
<iq type='get'
  from='device@example.org/device'
  to='provisioning.example.org'
  id='9'>
  <isFriend xmlns='urn:xmpp:iot:provisioning' jid='client1@example.org
      '/>
</iq>

<iq type='result'
  from='provisioning.example.org'
  to='device@example.org/device'
  id='9'>
  <isFriendResponse xmlns='urn:xmpp:iot:provisioning' jid='
      client1@example.org' result='true'/>
```

```
</iq>
```

Note: The provisioning server implicitly understands which two JIDs that are to be checked: The first one is the sender of the message, the second one is the JID available in the **jid** attribute in the request.
Note 2: Any resource information in the JID must be ignored by the provisioning server.

3.2.2 Friendship request rejected

The following diagram displays a friendship request from an external party being rejected as a result of the trusted third party negating the friendship:

The communication between the XMPP Device and the Provisioning Server could be as follows:

Listing 11: Friendship request rejected

```
<iq type='get'
  from='device@example.org/device'
  to='provisioning.example.org'
  id='10'>
  <isFriend xmlns='urn:xmpp:iot:provisioning' jid='client2@example.org
     '/>
</iq>

<iq type='result'
  from='provisioning.example.org'
  to='device@example.org/device'
  id='10'>
  <isFriendResponse xmlns='urn:xmpp:iot:provisioning' jid='
     client2@example.org' result='false'/>
</iq>
```

3.2.3 Unfriending existing friends

If the provisioning server decides that two friends in the network should no longer be friends and communicate with each other, it simply sends a message to at least one of the friends as follows:

The provisioning server should only send such messages to clients that have previously asked the provisioning server if friendship requests should be accepted or not.
Note: The device should only honor such messages, if the sender is the trusted third party. Such messages received from other entities not trusted should be silently ignored.

Listing 12: Unfriending existing friend

```
<message from='provisioning.example.org'
  to='device@example.org'>
  <unfriend xmlns='urn:xmpp:iot:provisioning' jid='client2@example.org
     '/>
</message>
```

3.2.4 Recommending friendships

The provisioning server can, apart from accepting new friendships and rejecting old friendships, also recommend new friendships. In this case, the provisioning server simply sends a message to one or both of the soon to be friends, as follows:

Listing 13: Recommending friendships

```
<message from='provisioning.example.org'
  to='device@example.org'>
  <friend xmlns='urn:xmpp:iot:provisioning' jid='client2@example.org'/
     >
</message>
```

Note that the receptor can still ask the provisioning server if it can form a friendship with the suggested friend, using the **isFriend** command.

3.3 Device Read-out

3.3.1 Rejecting read-outs

An important use case for provisioning in sensor networks is who gets to read out sensor data from which sensors. This use case details how communication with a provisioning server can help the device determine if a client has sufficient access rights to read the values of the device.

Note: This use case is an extension of the use case 'Read-out rejected' in the XEP-0323 Internet of Things - Sensor Data.

The following example shows the communication first between the client and the device, then between the device and the provisioning server, and last between the device and the client:

Listing 14: Rejecting read-outs

```
<iq type='get'
  from='master@example.org/amr'
  to='device@example.org'
  id='11'>
  <req xmlns='urn:xmpp:iot:sensordata' momentary='true' serviceToken='
      SERVICETOKEN1' userToken='USERTOKEN1' seqnr='1'/>
</iq>

<iq type='get'
  from='device@example.org/device'
  to='provisioning.example.org'
  id='12'>
  <canRead xmlns='urn:xmpp:iot:provisioning' jid='master@example.org'
      serviceToken='SERVICETOKEN1' userToken='USERTOKEN1' momentary='
      true'/>
</iq>

<iq type='result'
  from='provisioning.example.org'
  to='device@example.org/device'
  id='12'>
  <canReadResponse xmlns='urn:xmpp:iot:provisioning' jid='
      master@example.org' momentary='true' result='false'/>
</iq>

<iq type='error'
  from='device@example.org'
  to='master@example.org/amr'
  id='11'>
  <rejected xmlns='urn:xmpp:iot:sensordata' seqnr='1'>
    <error>Access denied.</error>
  </rejected>
</iq>
```

3.3.2 Restricting nodes during read-out

In case the device handles multiple nodes that can be read, the provisioning server has the possibility to grant read-out, but to limit the nodes that can be read out. The provisioning server does this by returning the list of nodes that can be read.

Note: This use case is an extension of the use case 'Read-out of multiple devices' in the XEP-0323 Internet of Things - Sensor Data.
Note 2: If the server responds, but without specifying a list of nodes, the device can assume that all nodes available in the original request are allowed to be read. If no nodes in the request are allowed to be read, the provisioning server must respond with a result='false', so the device can reject the read-out request.
The following example shows the communication first between the client and the device, then between the device and the provisioning server, and last between the device and the client:

Listing 15: Restricting nodes during read-out

```
<iq type='get'
  from='master@example.org/amr'
  to='device@example.org'
  id='13'>
  <req xmlns='urn:xmpp:iot:sensordata' momentary='true' serviceToken='
      SERVICETOKEN1' userToken='USERTOKEN1' seqnr='2'>
    <node nodeId='Device02'/>
    <node nodeId='Device03'/>
  </req>
</iq>

<iq type='get'
  from='device@example.org/device'
  to='provisioning.example.org'
  id='14'>
  <canRead xmlns='urn:xmpp:iot:provisioning' jid='master@example.org'
      momentary='true' serviceToken='SERVICETOKEN1' userToken='
      USERTOKEN1'>
    <node nodeId='Device02'/>
    <node nodeId='Device03'/>
  </canRead>
</iq>

<iq type='result'
  from='provisioning.example.org'
  to='device@example.org/device'
  id='14'>
  <canReadResponse xmlns='urn:xmpp:iot:provisioning' jid='
      master@example.org' momentary='true' result='true'>
    <node nodeId='Device02'/>
  </canReadResponse>
</iq>

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

```
  id='13'>
  <accepted xmlns='urn:xmpp:iot:sensordata' seqnr='2'/>
</iq>
```

Note that the provisioning server responds with a **canReadResponse** element, similar to the **canRead** element in the request, except only the nodes allowed to be read are read. The device must only permit read-out of nodes listed in the response from the provisioning server. Other nodes available in the request should be ignored.

3.3.3 Restricting fields during read-out

In case the provisioning server wants to limit the fields a device can send to a client, the provisioning server has the possibility to grant read-out, but list a set of fields the device is allowed to send to the corresponding client.

Note: If the server responds, but without specifying a list of field names, the device can assume that all fields available in the original request are allowed to be sent. If no fields in the request are allowed to be sent, the provisioning server must respond with a result='false', so the device can reject the read-out request.
The following example shows the communication first between the client and the device, then between the device and the provisioning server, and last between the device and the client:

Listing 16: Restricting fields during read-out

```
<iq type='get'
  from='master@example.org/amr'
  to='device@example.org'
  id='15'>
  <req xmlns='urn:xmpp:iot:sensordata' momentary='true' serviceToken='
      SERVICETOKEN1' userToken='USERTOKEN1' seqnr='3'/>
</iq>

<iq type='get'
  from='device@example.org/device'
  to='provisioning.example.org'
  id='16'>
  <canRead xmlns='urn:xmpp:iot:provisioning' jid='master@example.org'
      momentary='true' serviceToken='SERVICETOKEN1' userToken='
      USERTOKEN1'/>
</iq>

<iq type='result'
```

```
  from='provisioning.example.org'
  to='device@example.org/device'
  id='16'>
  <canReadResponse xmlns='urn:xmpp:iot:provisioning' jid='
      master@example.org' momentary='true' result='true'>
    <field name='Energy'/>
    <field name='Power'/>
  </canReadResponse>
</iq>

<iq type='result'
  from='device@example.org'
  to='master@example.org/amr'
  id='15'>
  <accepted xmlns='urn:xmpp:iot:sensordata' seqnr='3'/>
</iq>
```

Note that the provisioning server responds with a **canReadResponse** element, similar to the **canRead** element in the request, except only the fields allowed to be sent are listed. The client must only send fields having field names in this list.
Also note, that the provisioning server can return a list of both allowed nodes and allowed field names in the response. In this case, the device must only send allowed fields from allowed nodes, and ignore all other fields and/or nodes.

3.4 Device Control

3.4.1 Rejecting control actions

An important use case for provisioning in sensor networks is who gets to control devices, and what they can control. This use case details how communication with a provisioning server can help the device determine if a client has sufficient access rights to perform control actions on the device.

The following example shows the communication first between the client and the device, then between the device and the provisioning server, and last between the device and the client:

Listing 17: Rejecting control action

```
<iq type='set'
  from='master@example.org/amr'
  to='device@example.org'
  id='17'>
```

```
  <set xmlns='urn:xmpp:iot:control' xml:lang='en'>
    <boolean name='Output' value='true'/>
  </set>
</iq>

<iq type='get'
  from='device@example.org/device'
  to='provisioning.example.org'
  id='18'>
  <canControl xmlns='urn:xmpp:iot:provisioning' jid='master@example.
     org' serviceToken='SERVICETOKEN1' userToken='USERTOKEN1'>
    <parameter name='Output'/>
  </canControl>
</iq>

<iq type='result'
  from='provisioning.example.org'
  to='device@example.org/device'
  id='18'>
  <canControlResponse xmlns='urn:xmpp:iot:provisioning' jid='
     master@example.org' result='false'/>
</iq>

<iq type='error'
  from='device@example.org'
  to='master@example.org/amr'
  id='17'>
  <setResponse xmlns='urn:xmpp:iot:control' responseCode='
     InsufficientPrivileges'/>
</iq>
```

3.4.2 Restricting nodes during control

In case the device handles multiple nodes that can be read, the provisioning server has the possibility to grant control access, but to limit the nodes that can be controlled. The provisioning server does this by returning the list of nodes that can be controlled.

Note: If the server responds, but without specifying a list of nodes, the device can assume that all nodes available in the original request are allowed to be controlled. If no nodes in the request are allowed to be controlled, the provisioning server must respond with a result='false', so the device can reject the read-out request. The same is true for parameters: If the provisioning server does not specify parameters in the response, the caller can assume all parameters are allowed.
The following example shows the communication first between the client and the device,

then between the device and the provisioning server, and last between the device and the client:

Listing 18: Restricting nodes during control

```
<iq type='set'
  from='master@example.org/amr'
  to='concentrator@example.org'
  id='19'>
  <set xmlns='urn:xmpp:iot:control' xml:lang='en'>
    <node nodeId='DigitalOutput1'/>
    <node nodeId='DigitalOutput2'/>
    <node nodeId='DigitalOutput3'/>
    <node nodeId='DigitalOutput4'/>
    <boolean name='Output' value='true'/>
  </set>
</iq>

<iq type='get'
  from='concentrator@example.org/plc'
  to='provisioning.example.org'
  id='20'>
  <canControl xmlns='urn:xmpp:iot:provisioning' jid='master@example.
     org' serviceToken='SERVICETOKEN1' userToken='USERTOKEN1'>
    <node nodeId='DigitalOutput1'/>
    <node nodeId='DigitalOutput2'/>
    <node nodeId='DigitalOutput3'/>
    <node nodeId='DigitalOutput4'/>
    <parameter name='Output'/>
  </canControl>
</iq>

<iq type='result'
  from='provisioning.example.org'
  to='concentrator@example.org/plc'
  id='20'>
  <canControlResponse xmlns='urn:xmpp:iot:provisioning' jid='
     master@example.org' result='true'>
    <node nodeId='DigitalOutput2'/>
    <node nodeId='DigitalOutput3'/>
  </canControlResponse>
</iq>

<iq type='result'
  from='concentrator@example.org'
  to='master@example.org/amr'
  id='19'>
  <setResponse xmlns='urn:xmpp:iot:control' responseCode='OK'>
    <node nodeId='DigitalOutput2'/>
```

```
    <node nodeId='DigitalOutput3'/>
  </setResponse>
</iq>
```

Note that the provisioning server responds with a **canControlResponse** element, similar to the **canControl** element in the request, except only the nodes allowed to be controlled are included. The device must only permit control of nodes listed in the response from the provisioning server. Other nodes available in the request should be ignored.
Also note, that the restricted set of nodes and/or parameters returned from the provisioning server must be returned to the original caller, so it can act on the information that only a partial control action was allowed and taken.

3.4.3 Restricting parameters during control

In case the provisioning server wants to limit the control parameters a client can control in a device, the provisioning server has the possibility to grant control access, but list a set of parameters the client is allowed to control in the corresponding device.

Note: If the server responds, but without specifying a list of nodes, the device can assume that all nodes available in the original request are allowed to be controlled. If no nodes in the request are allowed to be controlled, the provisioning server must respond with a result='false', so the device can reject the read-out request. The same is true for parameters: If the provisioning server does not specify parameters in the response, the caller can assume all parameters are allowed.
The following example shows the communication first between the client and the device, then between the device and the provisioning server, and last between the device and the client:

Listing 19: Restricting parameters during control

```
<iq type='set'
  from='master@example.org/amr'
  to='plc@example.org'
  id='21'>
  <set xmlns='urn:xmpp:iot:control' xml:lang='en'>
    <boolean name='DigitalOutput1' value='true'/>
    <boolean name='DigitalOutput2' value='true'/>
    <boolean name='DigitalOutput3' value='true'/>
    <boolean name='DigitalOutput4' value='true'/>
    <int name='AnalogOutput1' value='65535'/>
    <int name='AnalogOutput2' value='65535'/>
    <int name='AnalogOutput3' value='65535'/>
```

```
    <int name='AnalogOutput4' value='65535'/>
  </set>
</iq>

<iq type='get'
  from='plc@example.org/plc'
  to='provisioning.example.org'
  id='22'>
  <canControl xmlns='urn:xmpp:iot:provisioning' jid='master@example.
     org' serviceToken='SERVICETOKEN1' userToken='user0001'>
    <parameter name='DigitalOutput1'/>
    <parameter name='DigitalOutput2'/>
    <parameter name='DigitalOutput3'/>
    <parameter name='DigitalOutput4'/>
    <parameter name='AnalogOutput1'/>
    <parameter name='AnalogOutput2'/>
    <parameter name='AnalogOutput3'/>
    <parameter name='AnalogOutput4'/>
  </canControl>
</iq>

<iq type='result'
  from='provisioning.example.org'
  to='plc@example.org/plc'
  id='22'>
  <canControlResponse xmlns='urn:xmpp:iot:provisioning' jid='
     master@example.org' result='true'>
    <parameter name='DigitalOutput1'/>
    <parameter name='DigitalOutput2'/>
    <parameter name='DigitalOutput3'/>
    <parameter name='DigitalOutput4'/>
  </canControlResponse>
</iq>

<iq type='result'
  from='plc@example.org'
  to='master@example.org/amr'
  id='21'>
  <setResponse xmlns='urn:xmpp:iot:control' responseCode='OK'>
    <parameter name='DigitalOutput1'/>
    <parameter name='DigitalOutput2'/>
    <parameter name='DigitalOutput3'/>
    <parameter name='DigitalOutput4'/>
  </setResponse>
</iq>
```

Note that the provisioning server responds with a **canControlResponse** element, similar to the **canControl** element in the request, except only the parameters allowed to be sent are listed. The device must only control parameters included in this list.

Also note, that the provisioning server can return a list of both allowed nodes and allowed parameter names in the response back to the client, so it can act on the information that only a partial control action was allowed and taken.

3.5 Cache

3.5.1 Clear cache

When the provisioning server updates access rights and user privileges in the system, it will send a **clearCache** command to corresponding devices. If a device was offline during the change, the provisioning server must send the **clearCache** message when the device comes online again. To acknowledge the receipt of the command, the client responds with a **clearCacheResponse** element. This response message does not contain any information on what was done by the client. It simply acknowledges the receipt of the command, to make sure the provisioning server does not resend the clear cache command again.
Note: The **clearCache** command does not include information on what has been changed, so the device needs to clear the entire cache. This to avoid complexities in making sure updates made to the provisioning rules works in all cases, and to minimize complexity in the implementation of the protocol on the sensor side. It is also not deemed to decrease network performance, since changing provisioning rules for a device is an exceptional event and therefore does not affect performance during normal operation.

Listing 20: Clear cache

```
<iq type='set'
  from='provisioning.example.org'
  to='device@example.org'
  id='23'>
  <clearCache xmlns='urn:xmpp:iot:provisioning'/>
</iq>

<iq type='result'
  from='device@example.org'
  to='provisioning.example.org'
  id='23'>
  <clearCacheResponse xmlns='urn:xmpp:iot:provisioning'/>
</iq>
```

3.6 Services

3.6.1 Getting a service token

A service requesting provisioning assistance, needs to retrieve a service token from the provisioning server, by providing a base-64 encoded X.509 certificate. The following example

shows how this can be done.

Listing 21: Requesting a service token

```
<iq type='get'
  from='device@example.org/device'
  to='provisioning.example.org'
  id='24'>
  <getToken xmlns='urn:xmpp:iot:provisioning'>BASE-64 ENCODED PUBLIC X
      .509 CERTIFICATE</getToken>
</iq>

<iq type='result'
  from='provisioning.example.org'
  to='device@example.org/device'
  id='24'>
  <getTokenChallenge xmlns='urn:xmpp:iot:provisioning' seqnr='1'>BASE
      -64 ENCODED CHALLENGE</getTokenChallenge>
</iq>

<iq type='get'
  from='device@example.org/device'
  to='provisioning.example.org'
  id='25'>
  <getTokenChallengeResponse xmlns='urn:xmpp:iot:provisioning' seqnr='
      1'>BASE-64 ENCODED RESPONSE</getTokenChallengeResponse>
</iq>

<iq type='result'
  from='provisioning.example.org'
  to='device@example.org/device'
  id='25'>
  <getTokenResponse xmlns='urn:xmpp:iot:provisioning' token='TOKEN'/>
</iq>
```

3.6.2 User access to service

Provisioning in sensor networks also requires control of user access to different services in the network. This use case shows how service access rights are controlled using a trusted provisioning server.

First, the user connects to the service in some way. This can be done using XMPP, HTTP, HTTPS or some other means. The service need to extract some form of identifying credentials from the user, and provide that to the provisioning server. The provisioning server

determines if the client has access rights to the service based on these credentials, and also provides the service with a **userToken** that the service can use in further communication with the provisioning server regarding the user and its privileges.
The following table lists some different types of credentials that the service can extract from the client:

Type	Protocols	Description
JID	XMPP	Allows provisioning to be done on the JID the user has.
IP Address	HTTP, HTTPS	Allows provisioning to be done on IP address or IP-ranges for instance.
Host Name	HTTP, HTTPS with DNS	If the service has access to the client host name, for instance in an intra network, this can be used for provisioning.
X.509 Certificate	HTTPS	If the client provides a client certificate, such a certificate can be used to provide provisioning.
X.509 Certificate Thumbprint	HTTPS	The client can also choose to validate the certificate itself and only send the certificate thumbprint to the provisioning server. Even though this provides somewhat lesser security than providing the entire certificate, it might be an option to distribute certificate validation checks across the network to lower the work load on the provisioning server.
User Name	HTTP, HTTPS, XMPP	If authenticated HTTP(S) access is implemented, the service can provide the user name as credentials. XMPP based clients can use information in the roster to provide user name information to the provisioning server.

Type	Protocols	Description
Geolocation	HTML5 over HTTP(S), XMPP	If the geographic location (longitude, latitude and possibly altitude) of the user client is known, it can be used. HTML 5 provides mechanism whereby the location of the client can be fetched. User Geolocation (XEP-0080) XEP-0080: User Geolocation <https://xmpp.org/extensions/xep-0080.html>. provides a mechanism whereby client location can be obtained over XMPP.
SSO Token	Intranet	If a single sign on token is available, such a token could be provided as credentials.
Protocol	Any	Connection protocol used to connect to the service.

The provisioning server receives these credentials, and decides if the user should have access to the service or not, based on rules configured in the provisioning service. If the user is granted access, a **userToken** is generated and returned to the service.
Now, the service can determine if this access grant is sufficient or not. It can require the user to login into the service first. If so, the service should provide the provisioning server with the user name used during login, when logged in.

Listing 22: User access to service

```
<!-{}- user connects to service -{}->

<iq type='get'
  from='service@example.org/service'
  to='provisioning.example.org'
  id='26'>
  <canAccess xmlns='urn:xmpp:iot:provisioning' serviceToken='
      SERVICETOKEN1'>
    <credentials type='IpAddress' value='10.0.0.1'/>
    <credentials type='Longitude' value='123.45'/>
    <credentials type='Latitude' value='67.89'/>
  </canAccess>
</iq>
```

```
<iq type='result'
  from='provisioning.example.org'
  to='service@example.org/service'
  id='26'>
  <canAccessResponse xmlns='urn:xmpp:iot:provisioning' userToken='
     USERTOKEN1' result='true'/>
</iq>

<!-{}- user performs login into service -{}->

<message from='service@example.org/service'
  to='provisioning.example.org'>
  <userLoggedIn xmlns='urn:xmpp:iot:provisioning' serviceToken='
     SERVICETOKEN1' userToken='USERTOKEN1' userName='Kermit' />
</message>

<!-{}- user continues interacting with service -{}->
```

3.7 User privileges

3.7.1 User privileges in service

When a user has been given access to a service, and properly been identified, the service can ask the provisioning service for detailed user privileges to control different aspects of the service. This can be done using the **hasPrivilege** command. Here, the service sends its **serviceToken** and the **userToken** earlier received when being granted access to the service. Furthermore a **privilegeId** has to be provided.
A **Privilege ID** is a string composed of one or a sequence of parts, delimited by period characters. The Privilege IDs form a tree of privileges, using an invisible, but common, root privilege.
The following table suggests some examples of Privilege IDs, with suggestive descriptions. (Only used as an example.)

Privilege ID	Description
Databases.Energy.Select	Gives the user the rights to select data from the Energy database.
Databases.Energy.Insert	Gives the user the rights to insert data into the Energy database.
Databases.Energy.Delete	Gives the user the rights to delete data from the Energy database.

Note: Note that privilege IDs are local to the service. Different services are allowed to use similar or same Privilege IDs, in different contexts and with different meanings. The provisioning server must separate Privilege IDs from different services.
The client must always provide full Privilege IDs to the provisioning server. The provisioning

server however, can grant partial privilege IDs, pointing to parent privilege nodes to a user or a role object, granting a user a specific role. If granting a parent privilege ID to a user or role, this is interpreted as giving the corresponding user or role the privileges of the entire sub-tree defined by the parent privilege ID.
If additional control over privileges is desired, negative privileges can be assigned to the user or role. Granted or explicitly rejected privileges are specified in a sequential list of full or partial privilege IDs. This list is then processed sequentially to determine if a privilege is granted or not.
Example: Consider the privileges above. Give a user or its corresponding role the following privileges in a sequential list:

- Exclude: Databases.Energy.Delete
- Include: Databases.Energy

This would give the user rights to select and insert data, but not to delete data from the Energy database.
Note: Care should be taken when constructing Privilege IDs, so they do not include variable data that potentially can create an infinite amount of privilege IDs. For example: Do not include user names, sensor IDs, etc., in privilege IDs, as the number of such entities cannot be estimated or is scalable beforehand.
The following diagram shows an example of how a service asks permission from a provisioning server before an action is taken:

Listing 23: User privileges in service

```
<!-{}- user wants to perform action -{}->

<iq type='get'
  from='service@example.org/service'
  to='provisioning.example.org'
  id='27'>
  <hasPrivilege xmlns='urn:xmpp:iot:provisioning' serviceToken='
      SERVICETOKEN1' userToken='USERTOKEN1' privilegeId='Sensors.View'
      />
</iq>

<iq type='result'
  from='provisioning.example.org'
  to='service@example.org/service'
  id='27'>
  <hasPrivilegeResponse xmlns='urn:xmpp:iot:provisioning' result='true
      '/>
```

```
</iq>

<!-{}- user performs action -{}->
```

3.7.2 Download all user privileges

To improve performance, services can download the entire set of user privileges, and perform privilege checks internally. The following diagram displays how the above two use cases could be handled in such a case:

Note: When downloading privileges using this command, a sequential list of full or partial privilege IDs will be returned together with the corresponding include or exclude flags. The above mentioned algorithm of determining user privileges must be implemented by the service, if this method is to be used.

Listing 24: Download all user privileges

```
<!-{}- user connects to service -{}->

<iq type='get'
  from='service@example.org/service'
  to='provisioning.example.org'
  id='28'>
  <canAccess xmlns='urn:xmpp:iot:provisioning' serviceToken='
      SERVICETOKEN1'>
    <credentials type='IpAddress' value='10.0.0.1'/>
    <credentials type='Longitude' value='123.45'/>
    <credentials type='Latitude' value='67.89'/>
  </canAccess>
</iq>

<iq type='result'
  from='provisioning.example.org'
  to='service@example.org/service'
  id='28'>
  <canAccessResponse xmlns='urn:xmpp:iot:provisioning' userToken='
      USERTOKEN1' result='true'/>
</iq>

<!-{}- user performs login into service -{}->

<message from='service@example.org/service'
  to='provisioning.example.org'>
  <userLoggedIn xmlns='urn:xmpp:iot:provisioning' serviceToken='
      SERVICETOKEN1' userToken='USERTOKEN1' userName='Kermit' />
```

```
</message>

<iq type='get'
  from='service@example.org/service'
  to='provisioning.example.org'
  id='29'>
  <downloadPrivileges xmlns='urn:xmpp:iot:provisioning' serviceToken='
      SERVICETOKEN1' userToken='USERTOKEN1'/>
</iq>

<iq type='result'
  from='provisioning.example.org'
  to='service@example.org/service'
  id='29'>
  <downloadPrivilegesResponse>
    <exclude id='Sensors.Delete'/>
    <include id='Sensors'/>
  </downloadPrivilegesResponse>
</iq>

<!-{}- user performs actions without interaction with the provisioning
    server -{}->
```

Note: If the user or service has not been correctly identified, logged in, etc., the resulting list must only include privileges that default users that are not logged in can have. This list can be empty.

4 Determining Support

If an entity is a Provisioning Server and supports the protocol specified herein, it MUST advertise that fact by returning a feature of "urn:xmpp:iot:provisioning" in response to Service Discovery (XEP-0030) [5] information requests.

Listing 25: 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 26: Service discovery information response

```
<iq type='result'
```

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

```
  from='provisioning.example.org'
  to='device@example.org/device'
  id='disco1'>
  <query xmlns='http://jabber.org/protocol/disco#info'>
    ...
    <feature var='urn:xmpp:iot:provisioning'/>
    ...
  </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) [6]. 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 JID vs Component Provisioning Servers

A client must treat the connection between a Provisioning Server differently if it is hosted as a client, having a JID, or if it is hosted as a Jabber Server Component. If it is hosted as a server component, there's no need for the thing to become friends with the Provisioning Server. Messages and requests can be made directly to the server component without having to add it to the roster or request presence subscriptions. If the Provisioning Server is hosted as a client, having a JID (@ in the address), the Provisioning Server must be added to the roster of the client before the client can communicate with the Provisioning Server.

5.2 Caching and cache time

To minimize network traffic, and optimize response time, devices should cache access rights and user privileges provided by the provisioning server. If memory is limited, items in the cache should be ordered by last access, and items with the oldest last access timestamp should be removed first. A safety valve can optionally be implemented as well, removing unused cache items after a certain age, even if memory is available.
The device can assume that access rights and user privileges on the provisioning server do not change over time, unless the provisioning server says so.
The provisioning server on the other hand, must keep track of when a device is online and offline, and clear the cache of the device if changes are made that affects the device. If the device was offline when those changes occurred, the provisioning server must send the clear cache command as soon as the device comes online again.
When creating a new trust relationship, the device should always clear its cache, if it contains information from before.

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

To minimize stress of the provisioning server during synchronous sensor start up, for instance after a power failure, all clients should aim to persist its cache if possible. Clients not persisting its cache may produce too much stress on the provisioning server on start-up, practically removing it from the network.

5.3 Working with multiple provisioning servers

When working with multiple provisioning servers, there are some things that should be considered:

- When a device requests information from the provisioning servers, this can be done in a parallel fashion. Even though the provisioning servers were added in a sequential fashion, there is no order or priority between the servers.
- When receiving different responses from different servers, with regards to access rights, privileges, relationships, etc., the union of rights, or the most forgiving or most accepting response should be used. If one of the servers grant a privilege, the privilege is assumed to exist. If one of the servers grant a friendship request, the friendship request should be granted, etc. Most probably, different servers will manage different subsets of entities on the network, and when they receive questions about unrecognized devices, they will simply deny access.
- If a provisioning server requires total control of the network, it should not allow secondary trust relationships. However, if such a server enters a network and there exist previous provisioning servers (i.e. they accept secondary trust relationships), more trust relationships are assumed to be acceptable.
- Even though examples in this document only list a secondary trust relationship, there is no such limit. There may exist many different trust relationships in a network or for a given device.

5.4 Automatic aggregation of services, users and privileges

One important design consideration when implementing a provisioning server is how to handle new services, users and privileges. One option might be to automatically ignore anything not recognized. Another option might be to dynamically add new services, user names and privileges to internal data sources, making it easier to manage new types of services dynamically. However, adding such items automatically might also make such data sources grow beyond control.

5.5 Reading devices from large subsystems

All examples in this document have been simplified examples where a few devices containing a few fields have been read. However, in many cases large subsystems with very many sensors

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

5.6 Different types of tokens

A small note regarding the use of different tokens. A service can get a **Service Token**, a device a **Device Token** and a user a **User Token**. When delegating these tokens to third parties, a service sends its **Service Token**. But, if the service does this within the context of a user action, the service sends both its **Service Token** and the users **User Token**. The same with a device. If a device delegates its token to a third party, it sends its **Device Token**. But if the device performs the action in the context of a user action, the device sends both its **Device Token** as well as its **User Token**.

6 Security Considerations

6.1 Trust Delegation

Delegating trust to a third party may create a weak link in the overall security of a sensor network. Therefore, it's vitally important that the following be adhered to:

- Trust should only be delegated in a secure way. Once delegated, it should be able to lock this trust so it cannot be changed without reverting the device to factory default settings. Such a locked delegation mode can be likened with a production mode, where vital configuration parameters should not be able to be changed.
- If allowing installation using a LED, as above, make sure the LED does not light up for a long time, limiting the window of access where the sensor can be linked to a provisioning server.
- Provisioning servers should be monitored during operation, since it provides a vital link in the operation of the network.
- Multiple provisioning servers could be allowed, for redundancy or for scalability. This specification does not limit the number of provisioning servers used in a network, not used by a device. Examples in this specification use only one provisioning server for simplicity.

- Generally, take care what kind of provisioning servers you allow in a network.

6.2 Token Challenges

When receiving token challenges from somebody, make sure you've sent the corresponding token to the corresponding party less than a minute before receiving the token challenge. If not, the token challenge might represent a malicious attempt by somebody else to use the token to gain privileges in the network otherwise not enjoyed.

7 IANA Considerations

This document requires no interaction with the Internet Assigned Numbers Authority (IANA) [7].

8 XMPP Registrar Considerations

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

9 XML Schema

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

  <xs:import namespace='urn:xmpp:iot:sensordata'/>

  <xs:element name='getToken' type='xs:base64Binary'/>
  <xs:element name='getTokenChallenge' type='GetTokenChallengeResponse
      '/>
  <xs:element name='getTokenChallengeResponse' type='
      GetTokenChallengeResponse'/>
  <xs:element name='getTokenResponse' type='GetTokenResponse'/>

  <xs:element name='tokenChallenge' type='TokenChallenge'/>
```

[7] 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/>.

```
<xs:element name='tokenChallengeResponse' type='xs:base64Binary'/>

<xs:element name='isFriend' type='Friend'/>
<xs:element name='isFriendResponse' type='FriendResponse'/>
<xs:element name='unfriend' type='Friend'/>
<xs:element name='friend' type='Friend'/>

<xs:element name='canRead' type='CanRead'/>
<xs:element name='canReadResponse' type='CanReadResponse'/>

<xs:element name='canControl' type='CanControl'/>
<xs:element name='canControlResponse' type='CanControlResponse'/>

<xs:element name='clearCache'>
  <xs:complexType/>
</xs:element>

<xs:element name='clearCacheResponse'>
  <xs:complexType/>
</xs:element>

<xs:element name='canAccess'>
  <xs:complexType>
    <xs:choice minOccurs='0' maxOccurs='unbounded'>
      <xs:element name='jid' type='JidCredential'/>
      <xs:element name='ip4' type='Ip4Credential'/>
      <xs:element name='ip6' type='Ip6Credential'/>
      <xs:element name='hostName' type='HostNameCredential'/>
      <xs:element name='x509Certificate' type='
          X509CertificateCredential'/>
      <xs:element name='x509CertificateThumbprint' type='
          X509CertificateThumbprintCredential'/>
      <xs:element name='userName' type='UserNameCredential'/>
      <xs:element name='longitude' type='LongitudeCredential'/>
      <xs:element name='latitude' type='LatitudeCredential'/>
      <xs:element name='altitude' type='AltitudeCredential'/>
      <xs:element name='sso' type='SsoCredential'/>
      <xs:element name='protocol' type='ProtocolCredential'/>
    </xs:choice>
    <xs:attribute name='serviceToken' type='xs:string' use='required
        '/>
  </xs:complexType>
</xs:element>

<xs:element name='canAccessResponse'>
  <xs:complexType>
    <xs:attribute name='result' type='xs:boolean' use='required'/>
    <xs:attribute name='userToken' type='xs:string' use='optional'/>
  </xs:complexType>
```

```
</xs:element>

<xs:element name='userLoggedIn'>
  <xs:complexType>
    <xs:attribute name='serviceToken' type='xs:string' use='required
        '/>
    <xs:attribute name='userToken' type='xs:string' use='required'/>
    <xs:attribute name='userName' type='xs:string' use='required'/>
  </xs:complexType>
</xs:element>

<xs:element name='hasPrivilege'>
  <xs:complexType>
    <xs:attribute name='serviceToken' type='xs:string' use='required
        '/>
    <xs:attribute name='userToken' type='xs:string' use='required'/>
    <xs:attribute name='privilegeId' type='PrivilegeId' use='
        required'/>
  </xs:complexType>
</xs:element>

<xs:element name='hasPrivilegeResponse'>
  <xs:complexType>
    <xs:attribute name='result' type='xs:boolean' use='required'/>
  </xs:complexType>
</xs:element>

<xs:element name='downloadPrivileges'>
  <xs:complexType>
    <xs:attribute name='serviceToken' type='xs:string' use='required
        '/>
    <xs:attribute name='userToken' type='xs:string' use='required'/>
  </xs:complexType>
</xs:element>

<xs:element name='downloadPrivilegesResponse'>
  <xs:complexType>
    <xs:choice minOccurs='0' maxOccurs='unbounded'>
      <xs:element name='include' type='Privilege'/>
      <xs:element name='exclude' type='Privilege'/>
    </xs:choice>
  </xs:complexType>
</xs:element>

<xs:complexType name='GetTokenResponse'>
  <xs:attribute name='token' type='xs:string' use='required'/>
</xs:complexType>

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

```
    <xs:simpleContent>
      <xs:extension base='xs:base64Binary'>
        <xs:attribute name='seqnr' type='xs:int' use='required'/>
      </xs:extension>
    </xs:simpleContent>
  </xs:complexType>

  <xs:complexType name='TokenChallenge'>
    <xs:simpleContent>
      <xs:extension base='xs:base64Binary'>
        <xs:attribute name='token' type='xs:string' use='required'/>
      </xs:extension>
    </xs:simpleContent>
  </xs:complexType>

  <xs:complexType name='Friend'>
    <xs:attribute name='jid' type='xs:string' use='required'/>
  </xs:complexType>

  <xs:complexType name='FriendResponse'>
    <xs:complexContent>
      <xs:extension base='Friend'>
        <xs:attribute name='result' type='xs:boolean' use='required'/>
        <xs:attribute name='secondaryTrustAllowed' type='xs:boolean'
            use='optional' default='false'/>
      </xs:extension>
    </xs:complexContent>
  </xs:complexType>

  <xs:complexType name='CanReadBase' abstract='true'>
    <xs:choice minOccurs='0' maxOccurs='unbounded'>
      <xs:element name='node'>
        <xs:complexType>
          <xs:attribute name='nodeId' type='xs:string' use='required'/
              >
          <xs:attribute name='sourceId' type='xs:string' use='optional
              '/>
          <xs:attribute name='cacheType' type='xs:string' use='
              optional'/>
        </xs:complexType>
      </xs:element>
      <xs:element name='field'>
        <xs:complexType>
          <xs:attribute name='name' type='xs:string' use='required'/>
        </xs:complexType>
      </xs:element>
    </xs:choice>
    <xs:attributeGroup ref='sn:fieldTypes'/>
```

```
  <xs:attribute name='all' type='xs:boolean' use='optional' default=
      'false'/>
  <xs:attribute name='historical' type='xs:boolean' use='optional'
      default='false'/>
  <xs:attribute name='jid' type='xs:string' use='required'/>
</xs:complexType>

<xs:complexType name='CanRead'>
  <xs:complexContent>
    <xs:extension base='CanReadBase'>
      <xs:attributeGroup ref='tokens'/>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>

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

<xs:complexType name='CanReadResponse'>
  <xs:complexContent>
    <xs:extension base='CanReadBase'>
      <xs:attribute name='result' type='xs:boolean' use='required'/>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>

<xs:complexType name='CanControlBase' abstract='true'>
  <xs:choice minOccurs='0' maxOccurs='unbounded'>
    <xs:element name='node'>
      <xs:complexType>
        <xs:attribute name='nodeId' type='xs:string' use='required'/
            >
        <xs:attribute name='sourceId' type='xs:string' use='optional
            '/>
        <xs:attribute name='cacheType' type='xs:string' use='
            optional'/>
      </xs:complexType>
    </xs:element>
    <xs:element name='parameter'>
      <xs:complexType>
        <xs:attribute name='name' type='xs:string' use='required'/>
      </xs:complexType>
    </xs:element>
  </xs:choice>
  <xs:attribute name='jid' type='xs:string' use='required'/>
```

```
</xs:complexType>

<xs:complexType name='CanControl'>
  <xs:complexContent>
    <xs:extension base='CanControlBase'>
      <xs:attributeGroup ref='tokens'/>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>

<xs:complexType name='CanControlResponse'>
  <xs:complexContent>
    <xs:extension base='CanControlBase'>
      <xs:attribute name='result' type='xs:boolean' use='required'/>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>

<xs:complexType name='JidCredential'>
  <xs:attribute name='value' type='xs:string' use='required'/>
</xs:complexType>

<xs:complexType name='Ip4Credential'>
  <xs:attribute name='value' type='xs:string' use='required'/>
</xs:complexType>

<xs:complexType name='Ip6Credential'>
  <xs:attribute name='value' type='xs:string' use='required'/>
</xs:complexType>

<xs:complexType name='HostNameCredential'>
  <xs:attribute name='value' type='xs:string' use='required'/>
</xs:complexType>

<xs:complexType name='X509CertificateCredential'>
  <xs:attribute name='base64' type='xs:string' use='required'/>
</xs:complexType>

<xs:complexType name='X509CertificateThumbprintCredential'>
  <xs:attribute name='base64' type='xs:string' use='required'/>
</xs:complexType>

<xs:complexType name='UserNameCredential'>
  <xs:attribute name='value' type='xs:string' use='required'/>
</xs:complexType>

<xs:complexType name='LongitudeCredential'>
  <xs:attribute name='value' type='xs:double' use='required'/>
</xs:complexType>
```

```
  <xs:complexType name='LatitudeCredential'>
    <xs:attribute name='value' type='xs:double' use='required'/>
  </xs:complexType>

  <xs:complexType name='AltitudeCredential'>
    <xs:attribute name='value' type='xs:double' use='required'/>
  </xs:complexType>

  <xs:complexType name='SsoCredential'>
    <xs:attribute name='value' type='xs:string' use='required'/>
  </xs:complexType>

  <xs:complexType name='ProtocolCredential'>
    <xs:attribute name='value' type='ProtocolName' use='required'/>
  </xs:complexType>

  <xs:complexType name='Privilege'>
    <xs:attribute name='id' type='PrivilegeId' use='required'/>
  </xs:complexType>

  <xs:simpleType name='ProtocolName'>
    <xs:restriction base='xs:string'>
      <xs:enumeration value='XMPP'/>
      <xs:enumeration value='HTTP'/>
      <xs:enumeration value='HTTPS'/>
      <xs:enumeration value='TcpSocket'/>
      <xs:enumeration value='UdpSocket'/>
      <xs:enumeration value='Queue'/>
      <xs:enumeration value='Internal'/>
      <xs:enumeration value='Other'/>
    </xs:restriction>
  </xs:simpleType>

  <xs:simpleType name='PrivilegeId'>
    <xs:restriction base='xs:string'>
      <xs:pattern value='^[^.]+([.][^.]+)*$'/>
    </xs:restriction>
  </xs:simpleType>

</xs:schema>
```

10 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/.

11 Acknowledgements

Thanks to Joachim Lindborg, Karin Forsell, Tina Beckman and Teemu Väisänen for all valuable feedback.