This specification describes how EXI compression can be used in XMPP networks.
Legal

Copyright

This XMPP Extension Protocol is copyright © 1999 – 2020 by the XMPP Standards Foundation (XSF).

Permissions

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

Warranty

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

Liability

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

Conformance

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

1 Introduction 1

2 Use Cases 2

2.1 Two approaches to use EXI 2

2.2 Stream Compression 2

2.2.1 Invalid setup 3

2.2.2 Proposing compression parameters 3

2.2.3 Uploading new schema files 5

2.2.4 Uploading compressed schema files 7

2.2.5 Downloading new schema files on server 9

2.2.6 Accessing quick configurations 10

2.2.7 Quick configuration failure 11

2.2.8 Start compression 11

2.3 EXI-specific stream elements 12

2.3.1 streamStart 12

2.3.2 streamEnd 13

2.4 Alternative Transport Binding for EXI/XMPP over TCP 13

2.4.1 DNS SRV lookup 19

2.4.2 Fallback Process 20

2.4.3 Default Payload Encoding Options for EXI/XMPP 20

3 Implementation Notes 21

3.1 EXI Encoding Preprocessing and Postprocessing 21

3.2 EXI options 22

3.2.1 Evaluations of Options 23

3.3 Transmission of EXI bodies and Session-wide Buffers 24

3.4 Preserving prefixes 24

3.5 Networks containing clients having limited memory 25

3.6 Caching schema files 25

3.7 Uploading vs. Downloading schemas 25

3.8 Server decompression and recompression vs. binary forwarding 26

3.9 Snapshot Repository for Well-known Schemas 26

3.10 Generation of Canonic Schemas 26

3.11 Using EXI Option Documents for Shortcut Setup 27

3.11.1 Shortcut Setup for Alternative Transport Binding 27

3.12 XMPP Schema files and their hash values 28

3.13 Known problems with existing schemas 32

3.13.1 Patch to avoid UPA for streams.xsd 33

3.13.2 Patch for missing wildcards in extensible schemas 34

3.14 Default schema for alternative bindings 34

3.15 Reserved Schema IDs 34
1 Introduction

The Efficient XML Interchange (EXI) Format \(^1\) is an efficient way to compress XML documents and XML fragments. This document provides information on how EXI can be used in XMPP streams to efficiently compress data transmitted between the server and the client. For certain applications (like applications in sensor networks) EXI is a vital component, decreasing packet size enabling sensors with limited memory to communicate efficiently. The strong support in EXI for generating efficient stubcodes is also vital to build efficient code in constrained devices.

Activating EXI compression requires a handshake to take place prior, where the server and client agree on a set of parameters. Some of these parameters may increase the compression ratio, at the cost of processing power and readability. These parameters include:

- Schemas to use.
- EXI version number.
- Data alignment (bit-packed, byte-alignment, pre-compression).
- If EXI-compressed data should be further compressed using additional compression.
- Strict or loose adherence to schemas.
- If comments, processing instructions, dtd:s, prefixes, lexical values, etc. should be preserved.
- If self-contained elements should be allowed.
- Alternate data type representations for types values.
- Block size for EXI compression.
- Maximum string length of value content items in string tables.
- Value partition capacity.

These parameters will be discussed in greater depth in the following sections. There are also default values that can be used to commence evaluating EXI compression.

The single most important property to agree on however, is the set of schemas to use during EXI compression. EXI compresses XML much more efficiently if schemas exist describing the format of the expected XML. Since the server is not supposed to know all possible XML schemas, a mechanism is provided in this document whereby schemas can be interchanged, so that the server can adapt its compression to the needs of the client.

EXI can be used through two bindings:

\(^1\) Efficient XML Interchange (EXI) Format \(<\text{http://www.w3.org/TR/exi}/>.\)
• Normal XMPP Port
• Dedicated Binary EXI Port

Both will be described in turn, in the following sections.

2 Use Cases

2.1 Two approaches to use EXI

There are two ways to use EXI to make efficient XMPP communication. The first method describes how to activate EXI-compression using Stream Compression (XEP-0138) (XEP-0138). The second method describes an alternative binding. This method does not use Stream compression as defined in XEP-0138, rather it allows clients to connect to the server and start using EXI directly from the beginning.

2.2 Stream Compression

The following sections assume the client connects through the normal XMPP port, and starts communicating with the server using uncompressed XML fragments. When the client connects to the XMPP Server, it will receive a list of features supported by the server:

```
Listing 1: Search Features

<stream:features>
    <starttls xmlns='urn:ietf:params:xml:ns:xmpp-tls'/>
    <compression xmlns='http://jabber.org/features/compress'>
        <method>zlib</method>
        <method>lzw</method>
        <method>exi</method>
        <method>exi:54321</method>
    </compression>
</stream:features>
```

Support for EXI compression through the normal XMPP port is detected by the existence of the exi compression method in the features stanza. If a port (static or dynamic) is available for a dedicated binary EXI/XMPP binding, this can be detected by the existence of the exi:PORT compression method, where PORT is replaced by the port number used. More information about this alternative method is available in the Alternative Bindings section.

Note: If the client already knows the port number of the dedicated binary EXI/XMPP binding, it can connect there directly, without the need to check the server features using the normal XMPP port.

Following is a list of use cases displaying how the client can configure and activate EXI

compression on the current binding.

### 2.2.1 Invalid setup

If the client attempts to activate an EXI stream at this point, before the negotiation of EXI properties has been performed, the server must respond with a **setup-failed** response.

#### Listing 2: Invalid setup

```xml
<compress xmlns='http://jabber.org/protocol/compress'>
  <method>exi</method>
</compress>

<failure xmlns='http://jabber.org/protocol/compress'>
  <setup-failed/>
</failure>
```

### 2.2.2 Proposing compression parameters

When the client decides to activate EXI compression, it sends a **setup** stanza containing parameter proposals to the server as follows:

#### Listing 3: Proposing compression parameters

```xml
<setup xmlns='http://jabber.org/protocol/compress/exi' version='1'
  strict='true' blockSize='1024'
  valueMaxLength='32' valuePartitionCapacity='100'>
  <schema ns='http://www.w3.org/XML/1998/namespace' bytes='4726'
    md5Hash='2e2cf9072dc058dca41b7ee77a5cb54'/>
  <schema ns='http://etherx.jabber.org/streams' bytes='3450' md5Hash='68719b98725477c46a70958d1ea7c781'/>
  <schema ns='jabber:client' bytes='6968' md5Hash='5e2d5cf0506e3f16336d295093d666c4'/>
  <schema ns='jabber:server' bytes='6948' md5Hash='dd95bd3055dfdd69984ed427d6356e0'/>
  <schema ns='jabber:x:roster' bytes='1077' md5Hash='00cb233dee8391067559c5dpee04f3d'/>
  <schema ns='urn:ietf:params:xml:ns:xmpp-sasl' bytes='2769' md5Hash='fd9a83f5c75628486ce18c0eb3a35995'/>
  <schema ns='urn:ietf:params:xml:ns:xmpp-streams' bytes='3315'
    md5Hash='75cd95aeb9f1f66110c3ddcf00c9b8'/>
  <schema ns='urn:ietf:params:xml:ns:xmpp-tls' bytes='688' md5Hash='dc18bc4da356c7b6e7a6c52a4330825'/>
  <schema ns='urn:ietf:params:xml:ns:xmpp-stanzas' bytes='3133'
    md5Hash='1a8d2158842f9134dc497de64b10c3f'/>
  <schema ns='http://jabber.org/protocol/compress/exi' bytes='15094'
    md5Hash='8b8f91b95d9101f0781e0ba9b4e106be'/>
</setup>
```
Note: Schema files are identified using three properties: its **target namespace**, its **byte size** and its **MD5 hash**. The MD5 hash provides a way to detect small changes in the file, even if the byte size and namespace are the same.

It is important that the client specify not only application specific namespaces in this request, but also the versions of the schemas for the core XMPP protocol namespaces and the schema for the XML namespace, containing XML attributes.

Note: Hash values and byte sizes of known schemas at the time of writing, can be found [here](#). However, these values are informational only. It is recommended that the developer makes sure exactly what version of the schema to use, and calculate the hash for it correspondingly. Also, some changes to some schemas might be necessary, which will affect the hash values. For more information about this, see the information about known problems.

After receiving the request, the server responds with a **setupResponse** stanza containing the parameters it can accept, based on the initial values provided by the client. Any buffer sizes, etc., may have been changed, but only lowered, never raised.

### Listing 4: Unable to accommodate parameters

```xml
<setupResponse xmlns='http://jabber.org/protocol/compress/exi' version='1' strict='true'>
  blockSize='1024' valueMaxLength='32'
  valuePartitionCapacity='100'/>
```
2 USE CASES

Schema files that the server does not have (based on namespace, byte size and MD5 hash) are marked with the `missingSchema` element instead of the normal `schema` element. At this point the client can choose to abort the EXI enablement sequence if it cannot accommodate itself with the proposed parameter settings provided by the server. The XMPP session will continue to work in its current state. Aborting does not require taking further action from the client.

2.2.3 Uploading new schema files

If the server lacks information about a schema file, it is specified in the response through the `missingSchema` elements. At this point, the client can either choose to accept that these schema files are not available, making compression less efficient, or choose to upload the missing schema files to the server. Of course, uploading schema files would require the device to have sufficient buffers and memory to store and upload the schema files in the first place. (If it is not possible to upload the schema files, consideration should be given to installing the schema files manually at the server.)

To upload a schema file, the client simply sends the schema file using an `uploadSchema` element, as follows:

Listing 5: Uploading schema file

```xml
<uploadSchema xmlns='http://jabber.org/protocol/compress/exi'
    contentType='Text'>
    PD94bWwgdmVyc2lvbj0nMS4wJyBlbmNvZGluZz0nVVRGLTgnPz4NCjx4czpzY2hlbWENCiAgICB4
    bWxuczp4cz0naHR0cDovL3d3dy53My5vcmcvMjAwMS9YTUxTY2hlbWENCiAgICB4bWxuczp4cz0n
    ZXNwYWJsZS1cm46eG1wcDpzbjpwc92aXNpbc92aXNpbc92aXNpbc92aXNpbc92aXNpbc92aXNpbc92a
    aXNpbc92aXNpbc92aXNpbc92aXNpbc92aXNpbc92aXNpbc92aXNpbc92aXNpbc92aXNpbc92aXNpbc92aX
    aXNpbc92aXNpbc92aXNpbc92aXNpbc92aXNpbc92aXNpbc92aXNpbc92aXNpbc92aXNpbc92aXNpbc92aXNp
    bc92aXNpbc92aXNpbc92aXNpbc92aXNpbc92aXNpbc92aXNpbc92aXNpbc92aXNpbc92aXNpbc92aXNpbc92aX
    aXNpbc92aXNpbc92aXNpbc92aXNpbc92aXNpbc92aXNpbc92aXNpbc92aXNpbc92aXNpbc92aXNpbc92aXNp
    bc92aXNpbc92aXNpbc92aXNpbc92aXNpbc92aXNpbc92aXNpbc92aXNpbc92aXNpbc92aXNpbc92aXNpbc92aX
    aXNpbc92aXNpbc92aXNpbc92aXNpbc92aXNpbc92aXNpbc92aXNpbc92aXNpbc92aXNpbc92aXNpbc92aXNp
    bc92aXNpbc92aXNpbc92aXNpbc92aXNpbc92aXNpbc92aXNpbc92aXNpbc92aXNpbc92aXNpbc92aXNpbc92aX
    aXNpbc92aXNpbc92aXNpbc92aXNpbc92aXNpbc92aXNpbc92aXNpbc92aXNpbc92aXNpbc92aXNpbc92aXNp
    bc92aXNpbc92aXNpbc92aXNpbc92aXNpbc92aXNpbc92aXNpbc92aXNpbc92aXNpbc92aXNpbc92aXNpbc92aX
    aXNpbc92aXNpbc92aXNpbc92aXNpbc92aXNpbc92aXNpbc92aXNpbc92aXNpbc92aXNpbc92aXNpbc92aXNp
    bc92aXNpbc92aXNpbc92aXNpbc92aXNpbc92aXNpbc92aXNpbc92aXNpbc92aXNpbc92aXNpbc92aXNpbc92aX
    aXNpbc92aXNpbc92aXNpbc92aXNpbc92aXNpbc92aXNpbc92aXNpbc92aXNpbc92aXNpbc92aXNpbc92aXNp
    bc92aXNpbc92aXNpbc92aXNpbc92aXNpbc92aXNpbc92aXNpbc92aXNpbc92aXNpbc92aXNpbc92aXNpbc92aX
    aXNpbc92aXNpbc92aXNpbc92aXNpbc92aXNpbc92aXNpbc92aXNpbc92aXNpbc92aXNpbc92aXNpbc92aXNp
    bc92aXNpbc92aXNpbc92aXNpbc92aXNpbc92aXNpbc92aXNpbc92aXNpbc92aXNpbc92aXNpbc92aXNpbc92aX
    aXNpbc92aXNpbc92aXNpbc92aXNpbc92aXNpbc92aXNpbc92aXNpbc92aXNpbc92aXNpbc92aXNpbc92aXNp
    bc92aXNpbc92aXNpbc92aXNpbc92aXNpbc92aXNpbc92aXNpbc92aXNpbc92aXNpbc92aXNpbc92aXNpbc92aX
    aXNpbc92aXNpbc92aXNpbc92aXNpbc92aXNpbc92aXNpbc92aXNpbc92aXNpbc92aXNpbc92aXNpbc92aXNp
    bc92aXNpbc92aXNpbc92aXNpbc92aXNpbc92aXNpbc92aXNpbc92aXNpbc92aXNpbc92aXNpbc92aXNpbc92aX
    ...
The schema itself is sent using base64 encoding to the server. This is to make sure a binary exact copy is transferred, maintaining encoding, processing instructions, etc. The server then computes the **target namespace**, **byte size** and **MD5 Hash** from the sent schema file.

If the client desires, it can test the EXI setup again. This is optional, but can be used to test that uploading the schema files, and any new property values are accepted by the server.

**Listing 6: Testing newly uploaded schema files**

```
<setup xmlns='http://jabber.org/protocol/compress/exi' version='1'
strict='true' blockSize='1024'
valueMaxLength='32' valuePartitionCapacity='100'>
  <schema ns='http://www.w3.org/XML/1998/namespace' bytes='4726'
    md5Hash='2e2cf9072dc058dcda41b7ee77a5cb54'/>
  <schema ns='http://etherx.jabber.org/streams' bytes='3450'
    md5Hash='68719b98725477c46a70958d1ea7c781'/>
  <schema ns='jabber:client' bytes='6968'
    md5Hash='5e2d5cbf0506f16336d295093d66c4'/>
  <schema ns='jabber:server' bytes='6948'
    md5Hash='dd95bd3055dfdd69984ed427cd6356e0'/>
  <schema ns='jabber:x:roster' bytes='1077'
    md5Hash='00cb233dee83919067559c5dcee04f3d'/>
  <schema ns='urn:ietf:params:xml:ns:xmpp-sasl' bytes='2769'
    md5Hash='fd9a83f5c75628486ce18c0eb3a35995'/>
  <schema ns='urn:xmpp:streams' bytes='3315'
    md5Hash='75cd95aeab9f1fd66110c3ddcf00c9b8'/>
  <schema ns='urn:xmpp:iot:control' bytes='6293'
    md5Hash='447a7a6c52aa43330825'/>
  <schema ns='urn:xmpp:iot:sensordata' bytes='8092'
    md5Hash='49b101e7dea39ccc31340a3c7871c43'/>
  <schema ns='urn:xmpp:iot:interoperability' bytes='1275'
    md5Hash='5d39845a0082715ff8807691698353bb'/>
  <schema ns='urn:xmpp:iot:provisioning' bytes='6303'
    md5Hash='3ed5360bc17eadb2a8949498c9af3f0c'/>
</setup>
```

And the server should then respond:
Listing 7: Agreement between client and server

```xml
<setupResponse xmlns='http://jabber.org/protocol/compress/exi' version='1' strict='true' "
  blockSize='1024' valueMaxLength='32' "
  valuePartitionCapacity='100' agreement='true' "
  configurationId='c76ab4ec-4993-4285-8c7a-098060581bb8'>" 
  agreement='true' "
  configurationId='096ab4ec-4993-4285-8c7a-098060581bb8'>" 
  md5Hash='2e2cf9072dc058dca41b7ee77a5cb54'/>
<schema ns='http://www.w3.org/XML/1998/namespace' bytes='4726' "
  md5Hash='4726'/>
<schema ns='http://etherx.jabber.org/streams' bytes='3450' md5Hash='68719b98725477c46a70958d1ea7c781'/>
<schema ns='jabber:client' bytes='6968' md5Hash='5e2d5cbf0506e3f16336d295093d66c4'/>
<schema ns='jabber:server' bytes='6948' md5Hash='dd95bd3055dfdd69984ed427cd65660'/>
<schema ns='jabber:x:roster' bytes='1077' md5Hash='00cb233de8391067559c5dcee04f3d'/>
<schema ns='urn:ietf:params:xml:ns:xmpp-sasl' bytes='2769' md5Hash='fd9a83f5c75628486ce1c0eb3a35995'/>
<schema ns='urn:ietf:params:xml:ns:xmpp-streams' bytes='3315' md5Hash='75cd95abf9f6610c3ddcf00c9b8'/>
<schema ns='urn:ietf:params:xml:ns:xmpp-tls' bytes='688' md5Hash='dc18bc4da35bc1be7a6c52aa4330825'/>
<schema ns='urn:ietf:params:xml:ns:xmpp-stanzas' bytes='3133' md5Hash='1a8d2158842af91346c97de64b10c3f'/>
<schema ns='http://jabber.org/protocol/compress/exi' bytes='15094' md5Hash='8b8f91b95d9101f078e0ba9b4e106be'/>
<schema ns='urn:xmpp:iot:control' bytes='6293' md5Hash='74dce5230e8c8df8c4ed2c9e98495b'/>
<schema ns='urn:xmpp:iot:sensordata' bytes='8092' md5Hash='49b101e7dee39ccc31340a33c7871c43'/>
<schema ns='urn:xmpp:iot:interoperability' bytes='1275' md5Hash='5d39845a0892715f98807691698535bb'/>
<missingSchema ns='urn:xmpp:iot:provisioning' bytes='6303' md5Hash='3ed5360bc17eadb2a8949498c9af3f0c'/>
</setupResponse>
```

Note the `agreement` attribute in the response this time. The server must set this attribute to true if it agrees with the proposal from the client. The client in turn can check this attribute as a quick way to check if agreement exists. When the server is in agreement it must also return a Configuration ID in the `configurationId` attribute. This Configuration ID can be used later to quicker enter into EXI compressed mode.

### 2.2.4 Uploading compressed schema files

The `uploadSchema` command has an optional attribute called `contentType` that can be used to send different types of documents to the server. This is not a MIME content type, but an enumeration with the following options:
<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Text</td>
<td>The schema is sent as plain text, albeit base-64 encoded. If no encoding is provided in the XML header of the schema file, UTF-8 encoding is assumed. This is the default value.</td>
</tr>
<tr>
<td>ExiBody</td>
<td>The schema file is sent as an EXI compressed file, but only the body is sent. *</td>
</tr>
<tr>
<td>ExiDocument</td>
<td>The schema file is sent as an EXI compressed file. The entire file, including Exi header is provided. *</td>
</tr>
</tbody>
</table>

(*) These options assume the following set of default EXI options are used. It is assumed the XMPP server has more capabilities than the client, so the following set of options must be supported by the XMPP server. The schema files can be precompressed and stored as binary files on the client for easier transmission.

<table>
<thead>
<tr>
<th>Option</th>
<th>Default value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Version</td>
<td>1</td>
</tr>
<tr>
<td>alignment</td>
<td>bit-packed</td>
</tr>
<tr>
<td>compression</td>
<td>false</td>
</tr>
<tr>
<td>strict</td>
<td>false</td>
</tr>
<tr>
<td>fragment</td>
<td>false</td>
</tr>
<tr>
<td>preserve</td>
<td>all false, except preserve prefixes that must be true or schema negotiation may fail.</td>
</tr>
<tr>
<td>selfContained</td>
<td>false</td>
</tr>
<tr>
<td>datatypeRepresentationMap</td>
<td>No map</td>
</tr>
<tr>
<td>blockSize</td>
<td>N/A</td>
</tr>
<tr>
<td>valueMaxLength</td>
<td>unbounded</td>
</tr>
<tr>
<td>valuePartitionCapacity</td>
<td>unbounded</td>
</tr>
</tbody>
</table>

Since EXI compression does not preserve the exact binary representation of the schema file (for instance it doesn’t preserve white space), the server cannot correctly compute byte size and an MD5 hash for the file. Therefore, the client needs to provide this information in the `uploadSchema` command using the `bytes` and `md5Hash` attributes. They are mandatory in case EXI compressed schema files are uploaded to the server. Also note that the byte length and MD5 Hash should be computed on the original XML Schema file, not the compressed or decompressed version.
2.2.5 Downloading new schema files on server

As an alternative to uploading a schema file to the server, the client can ask the server to download a schema file by itself. This is done using the `downloadSchema` command, as follows:

```
<downloadSchema xmlns='http://jabber.org/protocol/compress/exi' url='http://schemavault.example.org/compress/sn/provisioning.xsd'/>
```

The server tries to download the schema by itself, and then computes the target namespace, byte size and MD5 Hash from the downloaded schema.

When the schema has been downloaded, the following successful download response is returned:

```
<downloadSchemaResponse xmlns='http://jabber.org/protocol/compress/exi' url='http://schemavault.example.org/compress/sn/provisioning.xsd' result='true'/>
```

If an HTTP error occurred while trying to download the schema, a response as follows is returned:

```
<downloadSchemaResponse xmlns='http://jabber.org/protocol/compress/exi' url='http://schemavault.example.org/compress/sn/provisioning.xsd' result='false'>
  <httpError code='404' message='NotFound'/>
</downloadSchemaResponse>
```

If the URL could not be resolved, the following response is returned:

```
<downloadSchemaResponse xmlns='http://jabber.org/protocol/compress/exi' url='urlk://example.org/schema.xsd' result='false'>
  <invalidUrl message='Unrecognized_schema.'/>
</downloadSchemaResponse>
```

If a timeout occurred during the download attempt, the following response is returned:

```
<downloadSchemaResponse xmlns='http://jabber.org/protocol/compress/exi' url='http://schemavault.example.org/compress/sn/provisioning.xsd' result='false'>
```

If the url points to something that is not a schema, the following response is returned:

**Listing 13: Invalid Content Type**

```
<downloadSchemaResponse xmlns='http://jabber.org/protocol/compress/exi' url='http://schemavault.example.org/compress/sn/provisioning.xsd' result='false'>
  <invalidContentType contentTypeReturned='text/html'/>
</downloadSchemaResponse>
```

If an error occurs that is unforeseen by this specification, the server can simply respond with a generic error message, as follows:

**Listing 14: Other types of errors**

```
<downloadSchemaResponse xmlns='http://jabber.org/protocol/compress/exi' url='http://schemavault.example.org/compress/sn/provisioning.xsd' result='false'>
  <error message='No_free_space_left'/>
</downloadSchemaResponse>
```

*Note:* Downloading a schema might download a version which does not correspond to the desired version of the schema. It might for instance have been updated. This means the `bytes` and `md5Hash` values corresponding to the downloaded file will not match the values expected by the client. Therefore, it’s in this case important the client checks that the server actually downloaded the version of the schema required by the client so it doesn’t assume the server uses the same version of the schema when in actuality it doesn’t.

### 2.2.6 Accessing quick configurations

Once an EXI setup has been accepted by the server, and agreement is reached, the server will provide the client with a quick Configuration ID through the `configurationId` attribute. This Configuration ID can be used by the client during successive connections to the server, to skip the larger part of the handshake, as is shown in the following example:

**Listing 15: Accessing quick configurations**

```
<setup xmlns='http://jabber.org/protocol/compress/exi' configurationId = 'c76ab4ec-4993-4285-8c7a-098060581bb8'/>
```

*Note:* the quick configuration includes all accepted schemas and all EXI options agreed upon during the session when the configuration ID was returned. The `configurationId` attribute MUST NOT be used together with other option attributes or schema definitions in the setup.
request.
If the configuration is still available on the server, the server responds:

Listing 16: Quick configuration accepted

```xml
<setupResponse xmlns='http://jabber.org/protocol/compress/exi'
agreement='true' configurationId='c76ab4ec-4993-4285-8c7a-098060581bb8'/>
```

Note that schemas or options are not mentioned explicitly when using this quick setup approach.

### 2.2.7 Quick configuration failure

If the server for some reason does not remember the specific configuration requested by the client (the client might have been disconnected for a long time), it responds in the following manner:

Listing 17: Quick configuration failure

```xml
<setupResponse xmlns='http://jabber.org/protocol/compress/exi'
agreement='false' configurationId='c76ab4ec-4993-4285-8c7a-098060581bb8'/>
```

The agreement attribute is optional, with a default value of false. So, if the attribute is omitted, the client must consider the agreement to be nonexistent. When no agreement is reached using the quick configuration approach, the client must restart the handshake and propose new compression parameters.

### 2.2.8 Start compression

When EXI option negotiation has been completed, the client can tell the server that it is ready to start compression. It does this using the normal `compress` stanza, as follows:

```xml
<compress xmlns='http://jabber.org/protocol/compress'>
  <method>exi</method>
</compress>
```

The server now has the necessary knowledge on how the EXI engine should be configured for the current session and it responds as follows:

Listing 18: Compression accepted

```xml
<compressed xmlns='http://jabber.org/protocol/compress'/>
```
When the client receives acknowledgement that the compression method has been accepted, it restarts the stream, as explained in XEP 0138, except that it must not resend the `<stream>` start element sequence. Similarly, the client must not send a `</stream>` element when closing the session. Instead, special streamStart and streamEnd elements are sent. More information about that later.

2.3 EXI-specific stream elements

Because EXI engines need to close all open XML elements before decompressing, it cannot start the stream by sending only an open `<stream>` element, and close the stream by sending a closing `</stream>` element. Instead separate streamStart and streamEnd elements have to be sent, allowing for similar semantics on the EXI-compressed channel, as described in the following subsections.

For clarity, examples in this section are displayed in XML for readability. But it is understood that the elements are sent using EXI compression and using the options defined during setup.

2.3.1 streamStart

The first thing the client needs to do, once it opens the new EXI-compressed connection, whether it be through the normal XMPP connection or through the alternative EXI-only binding, is to send a streamStart element. This element replaces the start stream tag normally sent.

Listing 19: Start of EXI-compressed stream

```xml
<exi:streamStart from='client@im.example.org'
    to='im.example.org'
    version='1.0'
    xml:lang='en'
    xmlns:exi='http://jabber.org/protocol/compress/exi'>
  <exi:xmlns prefix='{}' namespace='jabber:client'/>
  <exi:xmlns prefix='streams' namespace='http://etherx.jabber.org/streams'/>
  <exi:xmlns prefix='exi' namespace='http://jabber.org/protocol/compress/exi'/>
</exi:streamStart>
```

There’s a semantic difference between only writing an open XML element, and sending a closed XML element separately, and that is in the definition of XML namespaces. XML namespaces and their corresponding prefixes defined in the normal `<streams:stream>` element will be available to all child elements following in a normal XMPP stream. However, to be able to do the same in an EXP-compressed XMPP stream, you need to define the namespaces and prefixes separately. Furthermore, the EXI/XMPP layer needs to make these namespace and prefix-definitions available to all following elements sent on the stream. The empty prefix is
syononymous with the default namespace to use.

2.3.2 streamEnd

Before closing the connection, the client needs to send a `streamEnd` element. This element replaces the closing stream tag send normally.

Listing 20: End of EXI-compressed stream

```xml
<exi:streamEnd xmlns:exi='http://jabber.org/protocol/compress/exi'/>
```

2.4 Alternative Transport Binding for EXI/XMPP over TCP

Alternative binding for EXI/XMPP is suitable for use cases such as factory automation, smart grid appliances, and other embedded use of communications. It works best if clients are constrained and does not update its specification frequently. In addition, the network should allow clients and servers to use not well-known port because this communication involves alternative TCP port.

Typical steps of communication is as follows (based on RFC 6120 ³).

1. (Optional) A client (foo@example.net) try to resolve a server with alternative binding for EXI/XMPP with DNS SRV lookup (ex. _xmpp-bclient._tcp.example.net. IN SRV)

2. (Optional) A DNS server tells a set of DNS RR to notify a server accepts EXI/XMPP binding (ex. SRV 10 10 15222 srv.example.net.) (Optional: the DNS server may tell the version of the default schema supported by the server. Currently there is only one version and has no effect. For further discussion, see draft-doi-exi-messaging-requirement.

3. The client connects to srv.example.net. 15222 with TCP and the server accepts the connection.

4. The client sends out 'EXI Cookie' (e.g. '$EXI') and starts EXI stream with an EXI Header without any option document (implies default encoding parameters). It sends out EXI events corresponds to start tag of `<stream:stream>`. Following shows XML Equivalent and EXI events

Listing 21: XML equivalent of stream start element (Client to Server)

```xml
<?xml version="1.0"?>
```

5. The server responds with EXI stream, with EXI cookie, without EXI option, and with appropriate events for stream tag.

Listing 23: XML equivalent of stream start element (Server to Client)

```xml
<?xml version="1.0"?>
<exi:streamStart xmlns:exi='http://jabber.org/protocol/compress/exi' version="1.0" from="jabber.example.org" xml:lang="en"
 xmlns:xm='http://www.w3.org/XML/1998/namespace' >
<exi:xmlns prefix="stream" namespace="http://etherx.jabber.org/streams" />
<exi:xmlns prefix="" namespace="jabber:client" />
</exi:streamStart>
```

Listing 24: Actual EXI Stream with EXI Header

```
0000000 8a 02 8d cc ae 0c 2d ae 0d 8c
0000020 2c 4c 4c e1 45 60 20 02 44 48 48 44 3c
0000040 3a 01 02 53 7d 06 73 5c 65 2b 2d 83 0b 1b
0000060 2b c3 6b 65 2b 2d 83 0b 1b 2b 3c 6b 65
```
6. If client needs TLS or SASL negotiation, it should be done at this step. As specified in Section 4.3.3 of RFC6120, both parties MUST not send events corresponds to </stream:stream> tag. (e.g. exi:streamEnd element)

7. If client needs to use different encoding option or schema than the default encoding options or the default schema, then the client shall start schema negotiation. The streams with alternate options/schemas SHOULD NOT have an EXI Options document to indicate the parameter is negotiated via previous XMPP stream. For example, the client want to use MUC option (XEP-0045), the following communication will occur. First, client try to renegotiate XML schema used in the communication.

Listing 25: XML equivalent of setup element (Client to Server)

```xml
<?xml version="1.0"?>
<exi:setup xmlns:exi='http://jabber.org/protocol/compress/exi'>
  <exi:schema ns='http://jabber.org/protocol/muc' bytes='1322'
    md5Hash='853ad555f102bb2b71da9a2f2787f4f9' />
  <exi:schema ns='http://jabber.org/protocol/muc#owner' bytes='1284'
    md5Hash='6e4e2257c1a4ba937fbdf71664a7e793' />
</exi:setup>
```

Listing 26: Actual EXI Stream

```
0000000 7b 0a a0 a0 a0 a2 24 14 6a 69 4a 57 84 02 5a c4 b3 85
0000020 aa 4a 84 f1 1d 07 79 1e 92 06 87 47 47 03 a2 f2
0000040 f6 a6 16 26 26 57 22 e6 f7 26 72 f7 07 26 f7 46
0000060 f6 36 f6 c2 f6 d7 56 32 10 28 88 ce 23 84 22 9d
0000100 81 51 16 a4 8c ef 5b 5e 70 98 c4 51 dc 74 8c 99
0000120 a1 d1 d1 c0 e8 bc bd a9 85 89 89 95 c8 b9 bd c9
0000140 9c bd c1 c9 bd d1 bd 8d bd b0 bd b5 d5 8c 8d bd
0000160 dd b9 95 ca
0000164
```

8. In the response the server accepts schema change.

Listing 27: XML equivalent of setup element (Server to Client)

```xml
<?xml version="1.0"?>
<exi:setupResponse xmlns:exi='http://jabber.org/protocol/compress/exi'>
  <exi:schema ns='http://jabber.org/protocol/muc' bytes='1322'
    md5Hash='853ad555f102bb2b71da9a2f2787f4f9' />
  <exi:schema ns='http://jabber.org/protocol/muc#owner' bytes='1284'
    md5Hash='6e4e2257c1a4ba937fbdf71664a7e793' />
</exi:setupResponse>
```
Listing 28: Actual EXI Stream

0000000 7c 08 c9 8d cd 8c 58 58 58 98 cc 0b 58 4c 8d 4d
0000020 4b 4d 18 8e 58 8b 4e 0e 58 4c 8d 4d 18 d8 8e 0d
0000040 4e 4d 4d 4e 4d 8e 4c 72 a8 28 89 05 1a 9a 52 95
0000060 e1 00 96 b1 2c e1 6a 92 a1 3c 47 41 de 47 a4 81
0000100 a1 d1 d1 c0 e8 bc bd a9 85 89 95 c8 b9 bd c9
0000120 9c bd c1 c9 bd d1 bd 8d bd b0 bd b5 d5 8c 84 0a
0000140 22 33 88 e1 08 a7 6f 54 45 a9 23 3b d6 d7 9c 26
0000160 a1 d1 d1 c0 e8 bc bd a9 85 89 95 c8 b9 bd c9
0000180 d5 8c 84 0a
0000200 62 65 72 2e 6f 72 67 2f 70 72 6f 74 6f 6c 2f 6d 75 63 23 6f 77 6e 65 72 c0
0000220

Note that the server may deny the negotiation with agreement="false" setupResponse (example omitted).

9. Just after receiving the setupResponse, client re-opens the stream. The new stream should have EXI header and may have EXI option header with updated options.

Listing 29: XML equivalent of stream start element (Client to Server)

```xml
<?xml version="1.0"?>
<exi:streamStart xmlns:exi='http://jabber.org/protocol/compress/
exi' version='1.0' to="jabber.example.org" xml:lang='en'
xmlns:xml='http://www.w3.org/XML/1998/namespace'>
<exi:xmlns prefix="stream" namespace='http://etherx.jabber.org/
streams" />
<exi:xmlns prefix="xml" namespace='http://www.w3.org/XML/1998/
namespace" />
</exi:streamStart>
```

Listing 30: Actual EXI Stream

0000000 98 40 8c ad c0 28 d4 c2 c4 c4 ca e4 5c ca f0 c2
0000020 da e0 d8 ca 5c de e4 ce 00 20 04 89 a1 d1 d1 c0
0000040 e8 bc bd bd 95 d1 a1 95 c9 e0 b9 a9 85 89 95 95 c8
0000060 b9 bd c9 9c bd cd d1 a9 95 85 b5 cc 21 cd d1 c9
0000080 16 95 85 b4 07 b5 30 b1 31 32 b9 1d 31 b6 34 b2 b7
00000a0 3a 01 02 66 87 47 47 03 a2 f2 f7 77 77 72 e7 73
00000c0 32 e6 f7 26 72 f5 84 d4 c2 f3 13 93 93 82 f6 e6
00000e0 16 d6 57 37 06 16 36 50 57 86 d6 ca
0000174

10. In response, server re-opens the stream with exi:streamStart tag.
**2 USE CASES**

Listing 31: XML equivalent of stream start element (Server to Client)

```xml
<?xml version="1.0"?><!
configuration id="761aabc0-a255-4b9b-89a1-4cb859559691" />
<exi:streamStart xmlns:exi='http://jabber.org/protocol/compress/
exi' version="1.0" to="jabber.example.org" xml:lang="en"
<exi:xmlns prefix="stream" namespace="http://etherx.jabber.org/
streams" /> 
<exi:xmlns prefix="" namespace="jabber:client" /> 
namespace" /> </exi:streamStart>
```

Listing 32: Actual EXI Stream

```
00000000 98 02 8d 4c 2c 4c 4c ae 45 cc af 0c 2d ae 0d 8c
0000020 a5 cd ee 5e e2 08 ca dc 20 10 02 44 d0 e8 e8 e0
0000040 74 5e 5e ca e8 d0 ca e4 f0 5c 4d c4 c4 ca e4
0000060 5c de e4 ce 5e e6 e8 e4 ca c2 da e6 10 e6 e8 e4
0000100 ca c2 da 03 da 98 58 98 99 5c 8e 98 db 1a 59 5b
0000120 9d 00 81 33 43 a3 a3 81 d1 79 7b bb bb b9 73 b9
0000140 99 73 7b 93 39 7a c2 6a 61 79 89 c9 c9 c1 7b 73
0000160 0b 6b 2b 9b 83 0b 28 2b c3 6b 65
0000174
```

11. After an exi:streamStart from the server to the client, they can communicate with EXI stream. The first level element in conventional XMPP is encoded as root element of EXI message. For example, a client may send MUC query with EXI.

Listing 33: XML equivalent of MUC chat message (Client to Server)

```xml
<?xml version="1.0"?><iq xmlns="jabber:client" type="set" to="sensors@conference.example.org" id="ab26a" > 
<query xmlns="http://jabber.org/protocol/muc#owner">
<x xmlns="jabber:x:data" type="submit" />
</query>
</iq>
```

Listing 34: Actual EXI Stream with EXI Header

```
00000000 42 20 73 65 6e 73 6f 72 73 40 63 6f 6e 66 65 72
0000020 65 6e 63 65 2e 65 78 61 6d 70 6c 65 2e 6f 72 67
0000040 07 61 62 32 36 61 47 d9 5e 1a 50 1a 98 58 98 99
0000060 5c 8b 99 5e 18 5b 5c 1b 19 4b 9b dc 99 cb dc d9
0000100 5b 9c db dc a8
0000105
```
This message has a query element under muc#owner namespace. This is performed efficiently because this series of messages from the last streamStart element has been encoded with the set of schemas and the set includes schemas for MUC. Otherwise, the encoding will become 'built-in grammar' even if the encoder and the decoder uses non-strict schema-informed grammar. This is not possible if either encoder or decoder does not support built-in grammar or the stream uses strict schema-informed grammar. In such cases, the whole message that contains undefined element or attribute SHOULD be dropped.

12. The client and the server may end the stream with exi:streamEnd tag anytime.

Listing 35: XML equivalent of stream end element (Client to Server)
```xml
<?xml version="1.0"?>
<streamEnd xmlns='http://jabber.org/protocol/compress/exi' />
```

Listing 36: Actual EXI Stream
```
0000000 96
0000001
```

Exactly same message should be sent from the server to the client to close the opposite way of the stream (example omitted).

13. The client now can start a preconfigured EXI/XMPP stream with explicit EXI option with the encoding option and canonic schema used in the previous negotiated session (shortcut setup)

Listing 37: EXI Option Document used for Shortcut Setup (Client to Server)
```xml
<?xml version="1.0"?>
  <common>
    <schemaid>c:761aabc0-a255-4b9b-89a1-4cb859559691</schemaid>
  </common>
</header>
```

Listing 38: XML Equivalent of streamStart
```xml
<?xml version="1.0"?>
  <exi:xmns prefix="stream" namespace="http://etherx.jabber.org/streams" />
</exi:streamStart>
```
14. The server may respond with the given configuration ID. This can skip the configuration setup and the communication can continue with preconfigured schemas.

Listing 39: EXI Option Document used for Shortcut Setup (Server to Client)
```xml
<?xml version="1.0"?>
  <common>
    <schemaId>c:761aabc0-a255-4b9b-89a1-4cb859559691</schemaId>
  </common>
</header>
```

Listing 40: XML Equivalent of streamStart in response
```xml
<?xml version="1.0"?>
<exi:streamStart xmlns:exi='http://jabber.org/protocol/compress/exi' version="1.0" from="jabber.example.org" xml:lang="en"
xmlns:xml='http://www.w3.org/XML/1998/namespace' >
<exi:xmlns prefix="stream" namespace="http://etherx.jabber.org/streams" />
</exi:streamStart>
```

Or server may deny to start the new communication by just sending streamEnd tag with default encoding.

Listing 41: XML Equivalent of streamEnd to deny the configuration
```xml
<?xml version="1.0"?>
<streamEnd xmlns='http://jabber.org/protocol/compress/exi' />
```

### 2.4.1 DNS SRV lookup

Just same with section 3.2.1 of RFC6120. The service name MUST be 'xmpp-bclient' (for binary client-to-server connections) or 'xmpp-bserver' (for binary server-to-server connections).
2.4.2 Fallback Process

Fallback to well-known XMPP ports (5222, 5269) without doing SRV lookup is allowed. In this case, an initiating entity SHOULD give up connection if it receives non-EXI data (e.g. no EXI cookie and no distinguishing bit is set) and SHOULD NOT do automatic retry.

When an initiating entity tries to communicate with an XMPP server with EXI, it SHOULD start the stream with an EXI cookie ('$EXI') to avoid ambiguity.

Note: this expects an XMPP server shall return some error in plain XML if the server receives EXI.

2.4.3 Default Payload Encoding Options for EXI/XMPP

When no EXI encoding option is given, or not specified by given EXI encoding option while negotiation, followings are the default encoding options for EXI/XMPP messages.

- alignment: bit-packed
- compression: false
- strict: true
- fragment: false
- preserve: all false
- selfContained: false
- schemald: "urn:xmpp:exi:default"
- datatypeReresentationMap: not exist
- blockSize: N/A
- valueMaxLength: 64
- valuePartitionCapacity: 64

In addition, local value learning mechanism is disabled by default as described in section 4 of EXI Profile.

- localValuePartitions: 0
- maximumNumberOfBuiltInElementGrammars: 0 (means no grammar learning)
- maximumNumberOfBuiltInProductions: 0 (means no production learning)

Corresponding parameter for EXI Profile is as follows:
3 Implementation Notes

3.1 EXI Encoding Preprocessing and Postprocessing

This section describes a common view on programming model between EXI/XMPP servers and clients. This is just an example and an implementation may do it in different ways. However, messages exchanged as a result should be identical.

As shown in this document, XMPP documents represented by EXI is slightly different from documents represented in XML. For example, each stanzas and first level elements under <stream:stream> tags must be encoded as a standalone EXI body (document). If namespace prefixes are declared in the root element of XML process (<stream:stream> tag), the prefix should be recovered on decoded XML on the receiver side (EXI may or may not preserve prefix). Therefore, some preprocessing and postprocessing is required to make current XMPP implementations work properly over EXI/XMPP channel. An example of the configuration is described in the following figure.

In the example, the box XMPP is conventional XMPP process. [EXI] represents EXI processor (encoder and decoder). [Proc] means preprocessor and postprocessor. Following pre-process must be applied to XML streams from XMPP process before giving the stream to EXI processors.
3 IMPLEMENTATION NOTES

1. `<stream:stream>` start tag must be converted to standalone `<exi:streamStart>` element. All attributes defined in the input XML except namespace declarations should be copied as is. Namespace declarations and prefixes must be converted to `exi:xmlns` element under `exi:streamStart` element.

2. All first level elements under the root `<stream:stream>` element must be converted to standalone (root) element of corresponding stanza elements. If there are missing namespace declarations in the element, appropriate prefix declarations should be added to the element before giving the element to the EXI encoder as a document. Contents of the elements should not be modified.

3. `</stream:stream>` end tag must be converted to standalone `<exi:streamEnd>` element.

Following post-process must be applied to XML streams from EXI processors before giving the stream to XMPP process.

1. Received `<exi:streamStart>` element must be converted to `<stream:stream>` start tag with appropriate attributes and namespace declarations. Prefixies defined in the `<exi:streamStart>` content must be saved under the current connection context of post-processor.

2. Received EXI messages must be reproduced as elements under `<stream:stream>` root element. In addition, the namespace declarations in the element must be replaced to prefixes given in the streamStart tag content if there is a corresponding namespace declaration.

3. `<exi:streamEnd>` element must be converted to `</stream:stream>` end tag. The context created under the connection may be released.

3.2 EXI options

There are many EXI options in EXI Format. Followings are brief description for use in XMPP. The alignment option is to control how the values are encoded. The default option 'bit-packed' fits most of communication use cases. If TLS compression is used at the same time, pre-compression will make the best result.

The compression option is a Boolean to specify additional entropy-based compression is performed over EXI encoding. This option works best for larger documents. Usual use cases expected in XMPP/EXI may not give much additional compactness to messages.

The strict option is a Boolean to let EXI grammars (schema-informed grammars) work more strictly. With a strict schema-informed grammar, only valid data with the schema is allowed in streams. This makes best compactness of a grammar and messages without additional entropy-based compression. With nonstrict schema-informed grammar, derived elements and attributes could be encoded in the built-in grammar. The built-in grammar is a dynamic, self-learning grammar model that gives full flexibility, with larger message size.

The preserve option is a set of Booleans to alter some production of events. Usually, all false
3 IMPLEMENTATION NOTES

(no preservation of comments, processing instructions, DTDs, namespace declarations and its prefixes, and lexical values) makes no problem on XMPP communication. However, uploading a binary schema may require the schema encoded with preserved namespace prefixes. The prefixes are used in type definitions, and without prefix preservation binding between type definition references and actual definitions will be lost.

The **selfContained option** is a Boolean to enable self contained encoding of elements. Self contained element has no dependency to context (string table state in EXI processors) and can be copied to other context with the same grammar. However, self contained elements cannot use external string tables in EXI processors and may result larger size.

The **datatypeRepresentationMap option** can be used to modify how types encoded in EXI. If encoders and decoders have special encoding, it can be specified here. In most use cases in XMPP/EXI, this option will not be used.

The **blockSize option** specifies the block size used for EXI compression. It has no effect if compression is not used.

The **valueMaxLength option** specifies the maximum length of re-used string in a stream. Larger value makes more strings captured in string tables in EXI processors. This means more memory needs to be allocated to process a stream. Because some use cases, such as Internet of Things, expects constrained nodes in the network, default value specified in this XEP is very small (64 characters).

The **valuePartitionCapacity option** specifies how many strings should be kept in a string table in a time. Default value of this is also small in this XEP (64 strings).

To control buffer and string table life time, this XEP adds a new option: **sessionWideBuffers**. If set to true, all buffers, string tables, etc. will be maintained during the entire session. This may improve performance during time since strings can be omitted in the compressed binary stream, but it might also in some cases degrade performance since more options are available in the tables, requiring more bits to encode strings. The default value is false, meaning that buffers, string tables, etc., are cleared between each stanza. (This option is EXI/XMPP specific.)

3.2.1 Evaluations of Options

This section describes effects in message size for some options.

The option **sessionWideBuffers** must have large effect in XMPP communication, because many XIDs are re-used many time within a XMPP stream. If sessionWideBuffers option is enabled, most of XIDs can be encoded in short identifiers as it appears in some previous messages. Followings are preliminary evaluation of message size of XML (plain old XML), EXI with sessionWideBuffers=false, and EXI with sessionWideBuffers=true.

- **Plain Old XML (without XML declaration, whitespaces)**: 5011 bytes in 22 messages
- **EXI without sessionWideBuffers**: 1614 bytes in 22 messages
- **EXI with sessionWideBuffers**: 1458 bytes in 22 messages
3.3 Transmission of EXI bodies and Session-wide Buffers

The transmission of EXI-compressed stanzas takes the form of a sequence of EXI bodies. In order for the recipient to be able to correctly interpret these incoming EXI bodies, the sender is required to flush any pending bits at the end of the last End Document (ED) event for each stanza and then send any pending bytes available in the output buffer. Since this makes sure each EXI body starts at an even byte boundary, it permits the recipient to decompress the body into an XML stanza.

Therefore, each stanza sent on the stream, must be compressed separately, reusing the same options as used by the stream. (Options are not sent on the stream, only the generated EXI bodies).

Compression of the stanza must be done in document mode, not fragment mode, including the Start Document (SD) and End Document (ED) events. If there are unwritten bits pending after the last End Document (ED) event (after the end of the stanza), Zero-bits are written until a byte boundary is created. The receptor must ignore bits in the last byte after the last End Document event has been received.

During setup of the EXI compression engine, the client can choose if buffers are to be reused between stanzas, or cleared between each stanza. This is done using the EXI over XMPP specific option `sessionWideBuffers`, which is false by default, meaning buffers and string tables are cleared between each stanza.

There may be cases where maintaining buffers and string tables throughout the session is preferable. Since strings are already available in the buffers, they don’t need to be output in the stream the first time they appear in a stanza. However, the number of strings in tables increase, and so does the number of bits required to encode them. Depending on what type of communication is performed, this option might give better results one way or another. If the same type of message is always sent, maintaining string buffers may be more efficient. But if the client sends many many different types of messages, clearing buffers may be more efficient.

Note that the stream of EXI bodies is indefinite. It only stops when the session is closed, i.e. when the socket connection is dropped. Therefore, the buffers can grow indefinitely unless control is maintained on what types of messages are sent, their contents (specifically string values), and to whom they are sent (JIDs being strings). All string tables and buffers must be cleared when a connection is lost.

Note also that if you want the option to enter a session in the middle of the flow to listen to the communication, you need to clear tables and buffers between each stanza, or you will not be able to decode the binary stream appropriately.

3.4 Preserving prefixes

Normally, prefixes are not preserved during EXI compression and decompression. If the communicating parties (sending client, XMPP server(s) and receiving clients) interpret incoming stanzas and content according to namespace, this should be sufficient. However, some implementations do not check namespaces, but prefix names used. In such cases, all
communicating parties are required to enable the preserve prefixes option during negotiation.

**Note:** It is not sufficient that one party enable this option. Both sender and received are required to enable this option, or prefix names will be lost in the transmission. Note also, that preserving prefix names result in less efficient compression. Therefore, all clients implementing EXI compression should strive to parse incoming XML based on namespace, not prefix name.

### 3.5 Networks containing clients having limited memory

To successfully implement a network with clients having limited memory, such as sensor networks, care should be taken to make sure necessary schema files are preinstalled on the server, to avoid the necessity to upload schema files from the clients. Clients with limited memory might be unable to perform this task.

An alternative may be to install a richer client, that can upload the schema files to the server dynamically, and installing it into the network. Any client uploading a schema file, will make that schema file available for EXI compression to any other client in the network.

### 3.6 Caching schema files

Schema files uploaded to the server should be cached on the server in some kind of schema repository. If memory is limited on the server, schema files should be sorted by last access. Schema files with the oldest last access timestamp could be removed to maintain the cache within an approved cache size.

Note that schema files have three keys: Target namespace, byte size and MD5 Hash. Multiple versions of a schema file may exist (that is, with the same target namespace but different byte sizes or MD5 hash codes). Note also, that for any practical purpose, schema files can be stored using only the MD5 hash as a key, since it is highly improbable that two different schema files will have the same MD5 hash (unless consciously created that way). MD5 hash values are always in lower case.

### 3.7 Uploading vs. Downloading schemas

When the server lacks information about a given XML schema, the client has two options for updating the server. Either it uploads the schema, or it asks the server to download one.

Uploading a schema has the advantage, that the client knows exactly the version that the server requires. It has the disadvantage, that the client needs to store the schema and send a possible large schema to the server. If EXI is used because the device has limited memory, uploading a schema might not be an option.

Downloading a schema has the advantage, that size of schema does not matter. The disadvantage is that asynchronous errors might occur, so the client needs to pay attention to the
responses returned by the server when downloading schemas. Also, downloading a schema, might download a version which does not correspond to the desired version of the schema. So, it’s more important in this case that the client checks that the server actually has the version of the schema required by the client.

3.8 Server decompression and recompression vs. binary forwarding

If two XMPP clients communicate with each other through an XMPP server, and both clients use EXI compression, the server must only forward binary packets if both EXI compressed channels have exactly the same setup. If any parameter is different, the server MUST always recompress packets sent through it. Since the server always needs to decompress incoming EXI compressed packets to decode headers, omitting the compression part might save the server some processing power, but not all. Note that, in some networks it might be common using similar compression settings, while in others different compression settings are most common. Also note that binary forwarding is only possible if session-wide buffers are not used.

3.9 Snapshot Repository for Well-known Schemas

Errata and other updates may happen to well-known schemas. Slightest modification to XML schemas may break interoperability of EXI nodes. However, negotiating everything is not efficient. Well-known and aged schemas that referred from the schema for EXI/XMPP shall be snapshoted for use in EXI processors. Actual snapshot is located in (TBD-URL).

3.10 Generation of Canonic Schemas

The interface between the XMPP engine, whether in the client or the server, and the EXI compression engine is required to provide the engine with ONE XML Schema to use during the compression. The Efficient XML Interchange (EXI) Format also specifies the use of Schema IDs identifying the schema to use. However, in the XMPP case, the schema to provide to the EXI compression engine must be created dynamically based on the handshake provided during setup of the connection. Since this generation must be done both on the server side as well as the client side, it is important that the schemas be created semantically equivalent. This section describes how to create such schemas, henceforth called Canonical Schemas is described in this section. A canonical schema is simply a wrapper importing each of the schemas negotiated for the connection. The schemas MUST be imported in ascending namespace order. After generating the canonical schema, it’s a good idea to create a corresponding Configuration ID. The Configuration ID however, includes not only the schemas imported into the canonical schema, but also the EXI options to use during compression/decompression. The
canonical schema should be persisted for simple reuse when quick setup is used.
The target namespace of the canonical schema MUST be urn:xmpp:exi:cs.
The Schema ID to use is irrelevant in the XMPP layer of communication. Therefore, the server
and client can create their own Schema IDs, according to some algorithm. It is not important
if the Schema IDs match, since they are not used in data transmitted between the client and
server.

3.11 Using EXI Option Documents for Shortcut Setup

The configurationId attribute has a similar attribute called configurationLocation. This
attribute provides a mechanism to setup an EXI connection rapidly using option documents
installed as files on the server or available on the network accessible through an URL.
A client may specify a configurationId or configurationLocation on exi:setup element.
If the server has corresponding setup configuration, the server may respond with an
exi:setupResponse with agreement="true". If the server does not know the configurationId
or does not be able to use the given configurationLocation, the server shall respond with an
exi:setupResponse with agreement="false".
This specification does not define the format of this Configuration Location attribute, and so
it is server specific, or if it is supported. If used on a server not supporting this attribute, or
if the contents of the attribute is invalid, the server returns an agreement=false response.
Otherwise the semantics of the configurationLocation attribute is the same as for the
configurationId attribute, except it provide a mechism for static configurations, while the
configurationId provides a mechanism for dynamic configurations.
The format for these option documents or locations is beyond the scope of this specification.
The format for these option documents or locations is beyond the scope of this specification.

3.11.1 Shortcut Setup for Alternative Transport Binding

With alternative transport bind, following rule for shortcut may be used, assuming a server
and a client have common shared configuration with configurationId="01234" as an example.

• Client can start configured stream with a exi:streamStart element encoded in the con-
figuration given in configurationId="01234". To indicate configurationId, a schemaid
corresponds to the configurationId prefixed by "c:" SHALL be used. In this example,
schemaid is "c:01234". EXI option in EXI option header other than the schemaid SHOULD
NOT be specified.

• If the server accepts the configuration, the server SHOULD respond with a 
exi:streamStart element encoded in the configuration given in the configurationId.
The server SHALL put the schemaid in the EXI header of the response. No EXI options
other than the schemaid SHOULD NOT be specified in the response EXI header. The EXI
header is the indication of configuration agreement and SHALL NOT be omitted.
• If the server does not accept the configuration, the server SHALL respond with an **exi:streamEnd** element encoded in the **default schema** and the EXI header option SHOULD be empty.

• The client can continue pre-configured stream if and only if it receives **exi:streamStart** element encoded in the configurationId with EXI option header that contains the identical schemaid. Otherwise, the client SHOULD start a new **exi:streamStart** with a new TCP connection an empty EXI option header (e.g. default schema). Current TCP connection SHOULD NOT be used for re-negotiation (configuration setup).

### 3.12 XMPP Schema files and their hash values

The following table lists XMPP schemas at the time of writing, and their corresponding bytes sizes and MD5 Hash values.

<table>
<thead>
<tr>
<th>Namespace</th>
<th>Schema</th>
<th>Bytes</th>
<th>MD5 hash</th>
</tr>
</thead>
<tbody>
<tr>
<td><a href="http://etherx.jabber.org/streams">http://etherx.jabber.org/streams</a></td>
<td>streams.xsd</td>
<td>3450</td>
<td>68719b98725747c46a70958d1ea7</td>
</tr>
<tr>
<td>path://jabber.org/features/amp</td>
<td>amp-feature.xsd</td>
<td>635</td>
<td>cb4f48c999c1c7ce15df4f3129e55f</td>
</tr>
<tr>
<td>path://jabber.org/features/compress</td>
<td>compress-feature.xsd</td>
<td>670</td>
<td>8ff631b7a9fa196470ec57897251</td>
</tr>
<tr>
<td>path://jabber.org/protocol/activity</td>
<td>activity.xsd</td>
<td>5043</td>
<td>b6168aacc7626a641d7f1f8b59c9...</td>
</tr>
<tr>
<td>path://jabber.org/protocol/address</td>
<td>address.xsd</td>
<td>1915</td>
<td>8770419083d7f044ee4b5e5e93e1</td>
</tr>
<tr>
<td>path://jabber.org/protocol/amp</td>
<td>amp.xsd</td>
<td>1858</td>
<td>b929b5024af9222cbf2f1af944e1ca</td>
</tr>
<tr>
<td>path://jabber.org/protocol/amp#errors</td>
<td>amp-errors.xsd</td>
<td>963</td>
<td>7a3666c21bbf2060f2658e9118bfc3</td>
</tr>
<tr>
<td>path://jabber.org/protocol/bytestreams</td>
<td>bytestreams.xsd</td>
<td>2813</td>
<td>83211fe7c25510d9254a9a66c3f31b</td>
</tr>
<tr>
<td>path://jabber.org/protocol/caps</td>
<td>caps.xsd</td>
<td>1072</td>
<td>ed68b9c089830155d1c5e9ad1113c1</td>
</tr>
<tr>
<td>path://jabber.org/protocol/chatstates</td>
<td>chatstates.xsd</td>
<td>272</td>
<td>c00838d97b9c32a37ac09813a275e</td>
</tr>
<tr>
<td>path://jabber.org/protocol/commands</td>
<td>commands.xsd</td>
<td>515</td>
<td>98b0e90b35e00b04b2253af26f4aa</td>
</tr>
<tr>
<td>path://jabber.org/protocol/compress</td>
<td>compress.xsd</td>
<td>505</td>
<td>7ac2a9dc472fa5f76be329b270b9</td>
</tr>
<tr>
<td>path://jabber.org/protocol/compress/exi</td>
<td>exi.xsd</td>
<td>15339</td>
<td>4c33b8ac3b902582f5e25233fd0</td>
</tr>
<tr>
<td>path://jabber.org/protocol/disco#info</td>
<td>disco-info.xsd</td>
<td>1783</td>
<td>d5e690d7aa80c7a75f47b40b0e2</td>
</tr>
<tr>
<td>path://jabber.org/protocol/disco#items</td>
<td>disco-items.xsd</td>
<td>1482</td>
<td>6010e2e5dfad587fc1609987805c7</td>
</tr>
<tr>
<td>path://jabber.org/protocol/feature-neg</td>
<td>feature-neg.xsd</td>
<td>766</td>
<td>a154eeb514f8acbe0291c48d4e73</td>
</tr>
<tr>
<td>path://jabber.org/protocol/geoloc</td>
<td>geoloc.xsd</td>
<td>2322</td>
<td>90b0df99a06ee77929955b63b3b7</td>
</tr>
<tr>
<td>path://jabber.org/protocol/http-auth</td>
<td>http-auth.xsd</td>
<td>1020</td>
<td>f3dd34da7014bf4d65909184d38</td>
</tr>
<tr>
<td>path://jabber.org/protocol/httpbind</td>
<td>httpbind.xsd</td>
<td>4028</td>
<td>59559ed65025c2e4ecf1a9ad675e</td>
</tr>
<tr>
<td>path://jabber.org/protocol/ibb</td>
<td>ibb.xsd</td>
<td>1557</td>
<td>45bac4486abb974f9d2e98d83e</td>
</tr>
<tr>
<td>path://jabber.org/protocol/mood</td>
<td>mood.xsd</td>
<td>3994</td>
<td>93bb8be5dab135b7189c475f3d80</td>
</tr>
<tr>
<td>path://jabber.org/protocol/muc</td>
<td>muc.xsd</td>
<td>1322</td>
<td>853ad555f102bb2b71da9a2f27b7</td>
</tr>
<tr>
<td>Namespace</td>
<td>Schema</td>
<td>Bytes</td>
<td>MD5 hash</td>
</tr>
<tr>
<td>-----------</td>
<td>--------</td>
<td>-------</td>
<td>----------</td>
</tr>
<tr>
<td>path://jabber.org/protocol/muc#admin</td>
<td>muc-admin.xsd</td>
<td>2260</td>
<td>627d39d909e18208f0b068ff3e1e4e</td>
</tr>
<tr>
<td>path://jabber.org/protocol/muc#owner</td>
<td>muc-owner.xsd</td>
<td>1284</td>
<td>6e4e2257c1a4ba937fbd7f17664a7d</td>
</tr>
<tr>
<td>path://jabber.org/protocol/muc#unique</td>
<td>muc-unique.xsd</td>
<td>512</td>
<td>43f77e28d574453f1208c653a4ebf</td>
</tr>
<tr>
<td>path://jabber.org/protocol/muc#user</td>
<td>muc-user.xsd</td>
<td>3881</td>
<td>4c2643e070a591d09dd4e1af15f8b</td>
</tr>
<tr>
<td>path://jabber.org/protocol/nick</td>
<td>nick.xsd</td>
<td>498</td>
<td>280dd5dac18c98559ea5148226d8</td>
</tr>
<tr>
<td>path://jabber.org/protocol/offline</td>
<td>offline.xsd</td>
<td>1157</td>
<td>e0c329c185c3339480a78b8662be4</td>
</tr>
<tr>
<td>path://jabber.org/protocol/pubsub</td>
<td>pubsub.xsd</td>
<td>7296</td>
<td>18b754fd463509c7c95bd5a6b5f44</td>
</tr>
<tr>
<td>path://jabber.org/protocol/pubsub#errors</td>
<td>pubsub-errors.xsd</td>
<td>4637</td>
<td>2047f8f150fffe1fa932da529528d898</td>
</tr>
<tr>
<td>path://jabber.org/protocol/pubsub#event</td>
<td>pubsub-event.xsd</td>
<td>4866</td>
<td>003ba8f4780822511a84cc8f83ca6a</td>
</tr>
<tr>
<td>path://jabber.org/protocol/pubsub#owner</td>
<td>pubsub-owner.xsd</td>
<td>4387</td>
<td>f0d2a4733fe7991135172ff6c8ce</td>
</tr>
<tr>
<td>path://jabber.org/protocol/rosterx</td>
<td>rosterx.xsd</td>
<td>1338</td>
<td>97bf4d958edef7b962eb8fe38c</td>
</tr>
<tr>
<td>path://jabber.org/protocol/rsm</td>
<td>rsm.xsd</td>
<td>1379</td>
<td>340cfd0827a128fba0fc6a28a916</td>
</tr>
<tr>
<td>path://jabber.org/protocol/shim</td>
<td>shim.xsd</td>
<td>911</td>
<td>989fc4e6a95c1e7763e1c17245a622</td>
</tr>
<tr>
<td>path://jabber.org/protocol/si</td>
<td>si.xsd</td>
<td>1530</td>
<td>d00cd67e00a6923ad0186c5f3707</td>
</tr>
<tr>
<td>path://jabber.org/protocol/si/profile/file-transfer</td>
<td>file-transfer.xsd</td>
<td>1479</td>
<td>8fbc4d2d7972696b30132a409233b</td>
</tr>
<tr>
<td>path://jabber.org/protocol/sipub</td>
<td>sipub.xsd</td>
<td>1655</td>
<td>3eafe26f5ade2fbb00cc9d3d54051e</td>
</tr>
<tr>
<td>path://jabber.org/protocol/soap#fault</td>
<td>soap-fault.xsd</td>
<td>856</td>
<td>df35f17931ab9d7cc747f89e7c84853</td>
</tr>
<tr>
<td>path://jabber.org/protocol/tune</td>
<td>tune.xsd</td>
<td>1237</td>
<td>d5ba7c1a0e61e7c2c1ele325f583e</td>
</tr>
<tr>
<td>path://jabber.org/protocol/waitinglist</td>
<td>waitinglist.xsd</td>
<td>2021</td>
<td>456376a80a3db8048e2eb421d514d</td>
</tr>
<tr>
<td>path://jabber.org/protocol/xdata-layout</td>
<td>xdata-layout.xsd</td>
<td>1874</td>
<td>c74f11654366f6a3935a2f95bc59</td>
</tr>
<tr>
<td>path://jabber.org/protocol/xdata-validate</td>
<td>xdata-validate.xsd</td>
<td>1402</td>
<td>a373bfb495c0563e86f20a8256c</td>
</tr>
<tr>
<td>path://www.w3.org/XML/1998/namespace</td>
<td>xml.xsd</td>
<td>4726</td>
<td>2e2cf9072dc0d58dca41bb7e77a56</td>
</tr>
<tr>
<td>jabber:client</td>
<td>client.xsd</td>
<td>6968</td>
<td>5e2d5e0f5063e1633ed295093d6</td>
</tr>
<tr>
<td>jabber:component:accept</td>
<td>accept.xsd</td>
<td>6635</td>
<td>eb052d6aad6e4a704728e63b9eb</td>
</tr>
<tr>
<td>jabber:component:connect</td>
<td>connect.xsd</td>
<td>6637</td>
<td>74497cf7c9a306dfd83d33647d16</td>
</tr>
<tr>
<td>jabber:iq:auth</td>
<td>iq-auth.xsd</td>
<td>1075</td>
<td>524d617af83f0eae26b92d032a1</td>
</tr>
<tr>
<td>jabber:iq:gateway</td>
<td>gateway.xsd</td>
<td>766</td>
<td>736df8b8976add3a4b605512d2d</td>
</tr>
<tr>
<td>jabber:iq:last</td>
<td>iq-last.xsd</td>
<td>700</td>
<td>93a082996e9a87209704502256b2</td>
</tr>
<tr>
<td>jabber:iq:oob</td>
<td>iq-oob.xsd</td>
<td>739</td>
<td>556fa1cd4d36ae1780ed2e0c153</td>
</tr>
<tr>
<td>Namespace</td>
<td>Schema</td>
<td>Bytes</td>
<td>MD5 hash</td>
</tr>
<tr>
<td>------------------------</td>
<td>--------------</td>
<td>-------</td>
<td>-----------------</td>
</tr>
<tr>
<td>jabber:iq:pass</td>
<td>iq-pass.xsd</td>
<td>1274</td>
<td>4cf2bab840ce9d592f573db2d1dd</td>
</tr>
<tr>
<td>jabber:iq:privacy</td>
<td>iq-privacy.xsd</td>
<td>3116</td>
<td>1defc6d0dbd1f625f46f3f68c188</td>
</tr>
<tr>
<td>jabber:iq:private</td>
<td>iq-private.xsd</td>
<td>605</td>
<td>b8d0aeb2370a2f8658304ba5767</td>
</tr>
<tr>
<td>jabber:iq:register</td>
<td>iq-register.xsd</td>
<td>2598</td>
<td>b79db0064b527f0da817e9033389</td>
</tr>
<tr>
<td>jabber:iq:roster</td>
<td>roster.xsd</td>
<td>1898</td>
<td>936bba0e8f386bc173735c3b416f6</td>
</tr>
<tr>
<td>jabber:iq:rpc</td>
<td>jabber-rpc.xsd</td>
<td>4853</td>
<td>351909f64a5a6b66c35972a53d497a</td>
</tr>
<tr>
<td>jabber:iq:search</td>
<td>iq-search.xsd</td>
<td>1574</td>
<td>1f202602a9b464d43a3e65791d8c5</td>
</tr>
<tr>
<td>jabber:iq:time</td>
<td>iq-time.xsd</td>
<td>933</td>
<td>dde555d5464f460b3aabb8948ad</td>
</tr>
<tr>
<td>jabber:iq:version</td>
<td>iq-version.xsd</td>
<td>763</td>
<td>22a89a8dfc2fb5471b0534b853330</td>
</tr>
<tr>
<td>jabber:server</td>
<td>server.xsd</td>
<td>6948</td>
<td>dd95bd3055df6694ed427cd6</td>
</tr>
<tr>
<td>jabber:server:dialback</td>
<td>dialback.xsd</td>
<td>1606</td>
<td>ddb71d38501ceea6d6f04b96aeb</td>
</tr>
<tr>
<td>jabber:x:conference</td>
<td>x-conference.xsd</td>
<td>1339</td>
<td>d06113e71790a8bf492fe3d0e76c</td>
</tr>
<tr>
<td>jabber:x:data</td>
<td>x-data.xsd</td>
<td>3562</td>
<td>ed9ac8c241c7f6503887c86b3d9e6</td>
</tr>
<tr>
<td>jabber:x:delay</td>
<td>x-delay.xsd</td>
<td>940</td>
<td>60bc644095747ef5e987845ce3a</td>
</tr>
<tr>
<td>jabber:x:encrypted</td>
<td>x-encrypted.xsd</td>
<td>469</td>
<td>95816171d561dad64d12112bb69a</td>
</tr>
<tr>
<td>jabber:x:event</td>
<td>x-event.xsd</td>
<td>1013</td>
<td>305f4be3c118b9d3d11d9aeea751</td>
</tr>
<tr>
<td>jabber:x:expire</td>
<td>x-expire.xsd</td>
<td>916</td>
<td>2c4f88b96a5a0a140532788aca47c62</td>
</tr>
<tr>
<td>jabber:x:oob</td>
<td>x-oob.xsd</td>
<td>668</td>
<td>d85e431c6308896f6e3139d947be</td>
</tr>
<tr>
<td>jabber:x:roster</td>
<td>x-roster.xsd</td>
<td>1077</td>
<td>c0b233dee83919067559c5dce06</td>
</tr>
<tr>
<td>jabber:x:signed</td>
<td>x-signed.xsd</td>
<td>463</td>
<td>979b3824f4bd9278f184f078e29</td>
</tr>
<tr>
<td>roster:delimiter</td>
<td>delimiter.xsd</td>
<td>3435</td>
<td>aad71def59561634db78a5c3aef0cad</td>
</tr>
<tr>
<td>storage:bookmarks</td>
<td>bookmarks.xsd</td>
<td>4488</td>
<td>a508bb2144a4d6e20671abe259</td>
</tr>
<tr>
<td>storage:rosternotes</td>
<td>rosternotes.xsd</td>
<td>445</td>
<td>2d5a4c3e97af7650a941b74473b1</td>
</tr>
<tr>
<td>urn:ietf:params:xml:ns:xmpp-bind</td>
<td>bind.xsd</td>
<td>852</td>
<td>e4f6b7c405e969b05e1df40c88f8bb5</td>
</tr>
<tr>
<td>urn:ietf:params:xml:ns:xmpp-e2e</td>
<td>e2e.xsd</td>
<td>598</td>
<td>da2132f0a69fc38968a57c8ef7fe66</td>
</tr>
<tr>
<td>urn:ietf:params:xml:ns:xmpp-sasl</td>
<td>sasl.xsd</td>
<td>2769</td>
<td>fd9a83f5c7562846ce18c0ebb3a55</td>
</tr>
<tr>
<td>urn:ietf:params:xml:ns:xmpp-session</td>
<td>session.xsd</td>
<td>444</td>
<td>cc1dad32ba05d184075979b7a1c98</td>
</tr>
<tr>
<td>urn:ietf:parameters:xml:ns:xmpp-stanzas</td>
<td>stanzaerror.xsd</td>
<td>3433</td>
<td>1a8d215b8424f9134dc947de464b10</td>
</tr>
<tr>
<td>urn:ietf:parameters:xml:ns:xmpp-streams</td>
<td>streamerror.xsd</td>
<td>345</td>
<td>75cd95aucb9f1fd66110cc3ddc0f0c</td>
</tr>
<tr>
<td>urn:ietf:parameters:xml:ns:xmpp-tls</td>
<td>tls.xsd</td>
<td>688</td>
<td>dc18bc4da35bc1be7a6c52a4333</td>
</tr>
<tr>
<td>Namespace</td>
<td>Schema</td>
<td>Bytes</td>
<td>MD5 hash</td>
</tr>
<tr>
<td>------------------------------</td>
<td>--------------</td>
<td>-------</td>
<td>---------------------------</td>
</tr>
<tr>
<td>urn:xmpp:archive</td>
<td>archive.xsd</td>
<td>10355</td>
<td>379f31dbd639e577cc3671989f896</td>
</tr>
<tr>
<td>urn:xmpp:attention:0</td>
<td>attention.xsd</td>
<td>621</td>
<td>42f9232182d298aa92f29d300d406</td>
</tr>
<tr>
<td>urn:xmpp:avatar:data</td>
<td>avatar-data.xsd</td>
<td>482</td>
<td>d705ada4740d78c5d404f470e61e</td>
</tr>
<tr>
<td>urn:xmpp:avatar:metadata</td>
<td>avatar-metadata.xsd</td>
<td>1782</td>
<td>d8c3b3f5805154161c81f73dc4fd</td>
</tr>
<tr>
<td>urn:xmpp:blocking</td>
<td>blocking.xsd</td>
<td>1437</td>
<td>f9540f863f741e300e4706c877f3d</td>
</tr>
<tr>
<td>urn:xmpp:blocking:errors</td>
<td>blocking-errors.xsd</td>
<td>627</td>
<td>28eb443b113cbe7147cfed90834a</td>
</tr>
<tr>
<td>urn:xmpp:bob</td>
<td>bob.xsd</td>
<td>846</td>
<td>990e71ba1e657f3a587cd60f0e758c</td>
</tr>
<tr>
<td>urn:xmpp:captcha</td>
<td>captcha.xsd</td>
<td>752</td>
<td>2128162211b39d19530769ccdec</td>
</tr>
<tr>
<td>urn:xmpp:delay</td>
<td>delay.xsd</td>
<td>762</td>
<td>34283385814c8db0dc3ad874ae57</td>
</tr>
<tr>
<td>urn:xmpp:eventlog</td>
<td>eventlogging.xsd</td>
<td>2021</td>
<td>b12ff6e4b79b1ddd7c06721a404</td>
</tr>
<tr>
<td>urn:xmpp:features:rosterver</td>
<td>versioning-feature.xsd</td>
<td>619</td>
<td>089d67a345ba701d21c1d3316a05</td>
</tr>
<tr>
<td>urn:xmpp:http</td>
<td>http-over-xmpp.xsd</td>
<td>5135</td>
<td>51c68b927a5cc0ab468ed081e10</td>
</tr>
<tr>
<td>urn:xmpp:iot:concentrators</td>
<td>sensor-network-concentrators.xsd</td>
<td>37801</td>
<td>0ea1b43d143b7857870e1397c93e</td>
</tr>
<tr>
<td>urn:xmpp:iot:control</td>
<td>sensor-network-control.xsd</td>
<td>6293</td>
<td>74dcea52300e8c8dfc4de2c9e904</td>
</tr>
<tr>
<td>urn:xmpp:iot:interoperability</td>
<td>xep-0000-IoT-Interoperability.xsd</td>
<td>1275</td>
<td>5d39845a0082715ff880769169835</td>
</tr>
<tr>
<td>urn:xmpp:iot:provisioning</td>
<td>sensor-network-provisioning.xsd</td>
<td>8856</td>
<td>3ed5360bc17eadb2a8949498c9a6f</td>
</tr>
<tr>
<td>urn:xmpp:iot:sensordata</td>
<td>sensor-data.xsd</td>
<td>8752</td>
<td>49b101e7dea39ccc31340a3c787</td>
</tr>
<tr>
<td>urn:xmpp:jingle:1</td>
<td>jingle.xsd</td>
<td>4926</td>
<td>29d29c6f994ce1b1c3d5d8c81b50</td>
</tr>
<tr>
<td>urn:xmpp:jingle:apps:rtp:1</td>
<td>jingle-apps-rtp.xsd</td>
<td>3481</td>
<td>3fd1ce6fe7fc1e2b3dd3c196fb44de9</td>
</tr>
<tr>
<td>urn:xmpp:jingle:apps:rtp:errors:1</td>
<td>jingle-apps-rtp-errors.xsd</td>
<td>705</td>
<td>bb476ac38da026742db94862f8d</td>
</tr>
<tr>
<td>urn:xmpp:jingle:apps:rtp:info:1</td>
<td>jingle-apps-rtp-info.xsd</td>
<td>1467</td>
<td>c8fc00ddc6f869f4a923c3979a9c6</td>
</tr>
<tr>
<td>urn:xmpp:jingle:apps:rtp:zrtp:1</td>
<td>jingle-apps-rtp-zrtp.xsd</td>
<td>732</td>
<td>9144b780fc33098ad0c5725394d5</td>
</tr>
</tbody>
</table>
### 3.13 Known problems with existing schemas

The following sections list some known problems that might affect already defined schemas. For these schemas to be used together with EXI compression, the recommended procedures...
should be considered. If changes to the schema is required, new byte sizes and hash values
must be computed for the changed schema correspondingly.
Patched version of XML schemas may be placed in the snapshot repository for well-known
schemas to avoid wild patches (and too many derived schema for the same model).

3.13.1 Patch to avoid UPA for streams.xsd

TBD: group discussion needed
streams.xsd defined in RFC6120 has 'unique particle attribution (UPA)' problem. UPA is
undeterministic attribution of an element between a wildcard and an explicit definition.
Details could be found in Appendix H of XML schema specification.
A simple way to solve UPA is to insert delimiters around wildcards to eliminate ambiguity.
A deterministic delimiter can resolve ambiguity. The other way to solve UPA is to use 'weak
wildcard model' introduced in XML schema 1.1. The cause of ambiguity is lack of precedence
between explicit definitions and wildcard definitions. As weak wildcards have weaker prece-
dence against explicit definitions, there are no ambiguity with UPA.
To keep the same semantics with current XML implementations, by default this proposal
recommends weak wildcard model on implementations. However, if an implementation does
not support weak wildcards, it may use streams.xsd with following patch applied.

```xml
--- streams.xsd 2013-03-25 17:37:52.733451313 +0900
+++ streams-UPA-patched.xsd 2013-03-25 17:57:44.889469868 +0900
@@ -32,10 +32,12 @@
   <xs:any namespace='urn:ietf:params:xml:ns:xmpp-sasl'
     minOccurs='0'
     maxOccurs='1' />
+   <xs:element name="delim" type="xs:boolean"
     minOccurs='1' maxOccurs='1' />
   <xs:element name="err:text"
     minOccurs='0'
     maxOccurs='1' />
```
4 SECURITY CONSIDERATIONS

maxOccurs='1'

3.1.3.2 Patch for missing wildcards in extensible schemas

3.14 Default schema for alternative bindings

Default EXI grammar used in EXI/XMPP SHALL be equivalent to the EXI grammar defined by the following schema. The default EXI grammar is used in initiating connection of EXI/XMPP alternate binding. The default schema is defined by following definition, and all the imported schemas SHOULD be same schemas described in the snapshot and SHOULD have the identical MD5 hash value described in this section.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema
 xmlns:xs="http://www.w3.org/2001/XMLSchema"
 xmlns:stream="http://etherx.jabber.org/streams"
 targetNamespace="urn:xmpp:exi:cs"
 elementFormDefault="qualified">
 <xs:import namespace="http://etherx.jabber.org/streams"
 schemaLocation="${snapshot_url}/streams.xsd" />
 <xs:import namespace="http://jabber.org/protocol/compress/exi"
 schemaLocation="${snapshot_url}/xep-0322-01.xsd" />
</xs:schema>
```

The ${snapshot-url} corresponds to yet-to-be-specified schema snapshot repository described in snapshot repository for well-known schemas section. The schemaId of this schema will is 'urn:xmpp:exi:default'. Schemald is ID for this instance and not a namespace identifier and intentionally different from the target namespace of the schema.

3.15 Reserved Schema IDs

Following schema IDs are reserved.

- Schema IDs starts with "c:". This is used as shortcut setup for alternative transport binding.
- Schema IDs starts with "urn:xmpp". This may be used to describe XEP-based schemas.

4 Security Considerations

Note that EXI compressed information, even though it is hard to decode by humans, is by no means encrypted. If sensitive data is to be sent over an EXI compressed channel, encryption should be considered as well.
The reason for using MD5 as a hashing mechanism to identify schema versions, is because MD5 has a small memory footprint and is easy to implement. However, it has a weakness: Given a hash value, it’s relatively easy to create another file returning the same hash value. However, it’s very difficult to create another file of the same size as the original, resulting in the same hash value. For this reason, file sizes are also included when identifying a schema. Since the security threat, and any possible consequences of somebody trying to inject invalid schemas to a server is relatively small, this is considered a sufficient protection against such threats.

The feature of uploading and downloading new schemas to the server is a feature that the server can disable for exceptional high security reasons, for instance, in high-security installations where total control of the domain is necessary. Clients should be aware if this fact and check setup after uploading or asking the server to download new schemas. If the setup fails a second time, i.e. schemas are still missing, the client MUST NOT try to upload or download the missing schemas again, since this would provoke an indefinite loop.

5 IANA Considerations

This XEP requires new service names (proposed ‘xmpp-bclient’ and ‘xmpp-bserver’) for SRV records registered in IANA. A static port number for a dedicated binary EXI/XMPP binding may also be requested.

6 XMPP Registrar Considerations

Some of schemas should be kept ‘as-is’ format in XSF registry. EXI interoperability requires schema stability, so a snapshot repository of version-controlled well-known schemas is requested to XMPP Registrar. Related discussion is in here. Also, patched versions such as described in Known problems should be placed in the same snapshot repository.

Furthermore, it is suggested that the table at http://xmpp.org/resources/schemas/ is updated with two columns containing updated byte size and MD5 hash information for the current files. Tools can be provided by the authors to automate the extraction of byte size and MD5 Hash information from the collection of schema files and transform them to appropriate formats.

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

The target namespace for canonical schemas urn:xmpp:exi:cs, for default schemas urn:xmpp:exi:default, as well as the EXI compress schema namespace http://jabber.org/protocol/compress/exi need to be registered on the list of known XSF schemas.
7 XML Schema

```xml
<?xml version='1.0' encoding='UTF-8'?>
<xs:schema
    xmlns:xs='http://www.w3.org/2001/XMLSchema'
    targetNamespace='http://jabber.org/protocol/compress/exi'
    xmlns='http://jabber.org/protocol/compress/exi'
    xmlns:xml='http://www.w3.org/XML/1998/namespace'
    xmlns:client='jabber:client'
    xmlns:streams='http://etherx.jabber.org/streams'
    elementFormDefault='qualified'>

    <xs:import namespace='http://etherx.jabber.org/streams'/>

    <xs:element name='setup' type='Setup'/>
    <xs:element name='setupResponse' type='SetupResponse'/>

    <xs:complexType name='Setup'>
        <xs:choice minOccurs='0' maxOccurs='unbounded'>
            <xs:element name='schema' type='Schema'/>
            <xs:element name='datatypeRepresentationMap' type='DatatypeRepresentationMap'/>
        </xs:choice>
        <xs:attributeGroup ref='Options'/>
        <xs:attribute name='configurationId' type='xs:string' use='optional'/>
        <xs:attribute name='configurationLocation' type='xs:string' use='optional'/>
    </xs:complexType>

    <xs:complexType name='SetupResponse'>
        <xs:choice minOccurs='0' maxOccurs='unbounded'>
            <xs:element name='schema' type='Schema'/>
            <xs:element name='datatypeRepresentationMap' type='DatatypeRepresentationMap'/>
            <xs:element name='missingSchema' type='Schema'/>
        </xs:choice>
        <xs:attributeGroup ref='Options'/>
        <xs:attribute name='configurationId' type='xs:string' use='optional'/>
        <xs:attribute name='configurationLocation' type='xs:string' use='optional'/>
        <xs:attribute name='agreement' type='xs:boolean' use='optional'
                     default='false'/>
    </xs:complexType>

    <xs:complexType name='Schema'>
        <xs:attribute name='ns' type='xs:string' use='required'/>
    </xs:complexType>

</xs:schema>
```
<xs:attribute name='bytes' type='xs:positiveInteger' use='required'/>
<xs:attribute name='md5Hash' type='MD5Hash' use='required'/>
</xs:complexType>

<xs:complexType name='DatatypeRepresentationMap'>
  <xs:attribute name='type' type='xs:string' use='required'/>
  <xs:attribute name='representAs' type='xs:string' use='required'/>
</xs:complexType>

<xs:attributeGroup name='Options'>
  <xs:attribute name='version' type='xs:positiveInteger' use='optional' default='1'/>
  <xs:attribute name='alignment' type='Alignment' use='optional' default='bit-packed'/>
</xs:attributeGroup>

<xs:annotation>
  <xs:documentation>The alignment option is used to control the alignment of event codes and content items. The value is one of bit-packed, byte-alignment or pre-compression, of which bit-packed is the default value assumed when the "alignment" element is absent in the EXI Options document. The option values byte-alignment and pre-compression are effected when "byte" and "pre-compress" elements are present in the EXI Options document, respectively. When the value of compression option is set to true, alignment of the EXI Body is governed by the rules specified in 9. EXI Compression instead of the alignment option value. The "alignment" element MUST NOT appear in an EXI options document when the "compression" element is present.</xs:documentation>
</xs:annotation>

<xs:attribute name='compression' type='xs:boolean' use='optional' default='false'>
  <xs:annotation>
    <xs:documentation>The compression option is a Boolean used to increase compactness using additional computational resources. The default value "false" is assumed when the "compression" element is absent in the EXI Options document whereas its presence denotes the value "true". When set to true, the event codes and associated content are compressed according to 9. EXI Compression regardless of the alignment option value. As mentioned above, the "compression" element MUST NOT appear in an EXI options document when the "compression" element is present.</xs:documentation>
  </xs:annotation>
</xs:attribute>

alignment" element is present.</xs:documentation>
</xs:annotation>
</xs:attribute>
<xs:attribute name='strict' type='xs:boolean' use='optional' default='false'>
  <xs:annotation>
    <xs:documentation>The strict option is a Boolean used to increase compactness by using a strict interpretation of the schemas and omitting preservation of certain items, such as comments, processing instructions and namespace prefixes. The default value "false" is assumed when the "strict" element is absent in the EXI Options document whereas its presence denotes the value "true". When set to true, those productions that have NS, CM, PI, ER, and SC terminal symbols are omitted from the EXI grammars, and schema-informed element and type grammars are restricted to only permit items declared in the schemas. A note in section 8.5.4.4.2 Adding Productions when Strict is True describes some additional restrictions consequential of the use of this option. The "strict" element MUST NOT appear in an EXI options document when one of "dtd", "prefixes", "comments", "pis" or "selfContained" element is present in the same options document.</xs:documentation>
  </xs:annotation>
</xs:attribute>
<xs:attribute name='preserveComments' type='xs:boolean' use='optional' default='false'>
  <xs:annotation>
    <xs:documentation>Comments are preserved. Must not be used together with the strict option.</xs:documentation>
  </xs:annotation>
</xs:attribute>
<xs:attribute name='preservePIs' type='xs:boolean' use='optional' default='false'>
  <xs:annotation>
    <xs:documentation>Processing instructions are preserved. Must not be used together with the strict option.</xs:documentation>
  </xs:annotation>
</xs:attribute>
<xs:attribute name='preserveDTD' type='xs:boolean' use='optional' default='false'>
  <xs:annotation>
    <xs:documentation>DTD is preserved. Must not be used together with the strict option.</xs:documentation>
  </xs:annotation>
</xs:attribute>
<xs:attribute name='preservePrefixes' type='xs:boolean' use='optional' default='false'>
  <xs:annotation>
    <xs:documentation>Prefixes are preserved. Must not be used together with the strict option.</xs:documentation>
  </xs:annotation>
</xs:attribute>

<xs:attribute name='preserveLexical' type='xs:boolean' use='optional' default='false'>
  <xs:annotation>
    <xs:documentation>Lexical form of element and attribute values can be preserved in value content items. Can be used together with the strict option.</xs:documentation>
  </xs:annotation>
</xs:attribute>

<xs:attribute name='selfContained' type='xs:boolean' use='optional' default='false'>
  <xs:annotation>
    <xs:documentation>The selfContained option is a Boolean used to enable the use of self-contained elements in the EXI stream. Self-contained elements may be read independently from the rest of the EXI body, allowing them to be indexed for random access. The "selfContained" element MUST NOT appear in an EXI options document when one of "compression", "pre-compression" or "strict" elements are present in the same options document. The default value "false" is assumed when the "selfContained" element is absent from the EXI Options document whereas its presence denotes the value "true".</xs:documentation>
  </xs:annotation>
</xs:attribute>

<xs:attribute name='blockSize' type='xs:positiveInteger' use='optional' default='1000000'>
  <xs:annotation>
    <xs:documentation>The blockSize option specifies the block size used for EXI compression. When the "blockSize" element is absent in the EXI Options document, the default blocksize of 1,000,000 is used. The default blockSize is intentionally large but can be reduced for processing large documents on devices with limited memory.</xs:documentation>
  </xs:annotation>
</xs:attribute>
<xs:attribute name='valueMaxLength' type='xs:positiveInteger' use='optional'>
  <xs:annotation>
    <xs:documentation>The valueMaxLength option specifies the maximum length of value content items to be considered for addition to the string table. The default value "unbounded" is assumed when the "valueMaxLength" element is absent in the EXI Options document.</xs:documentation>
  </xs:annotation>
</xs:attribute>

<xs:attribute name='valuePartitionCapacity' type='xs:positiveInteger' use='optional'>
  <xs:annotation>
    <xs:documentation>The valuePartitionCapacity option specifies the maximum number of value content items in the string table at any given time. The default value "unbounded" is assumed when the "valuePartitionCapacity" element is absent in the EXI Options document. Section 7.3.3 Partitions Optimized for Frequent use of String Literals specifies the behavior of the string table when this capacity is reached.</xs:documentation>
  </xs:annotation>
</xs:attribute>

<xs:attribute name='sessionWideBuffers' type='xs:boolean' use='optional' default='false'>
  <xs:annotation>
    <xs:documentation>If set to true, all buffers, string tables, etc. will be maintained during the entire session. This may improve performance during time since strings can be omitted in the compressed binary stream, but it might also in some cases degrade performance since more options are available in the tables, requiring more bits to encode strings. The default value is false, meaning that buffers, string tables, etc., are cleared between each stanza.</xs:documentation>
  </xs:annotation>
</xs:attribute>

<xs:simpleType name='MD5Hash'>
  <xs:restriction base='xs:string'>
    <xs:pattern value='[0-9a-f]{32}' />
  </xs:restriction>
</xs:simpleType>
<xs:simpleType name='Alignment'>
  <xs:restriction base='xs:string'>
    <xs:enumeration value='bit-packed'>
      <xs:annotation>
        <xs:documentation>The alignment option value bit-packed indicates that the event codes and associated content are packed in bits without any padding in-between.</xs:documentation>
      </xs:annotation>
    </xs:enumeration>
    <xs:enumeration value='byte-alignment'>
      <xs:annotation>
        <xs:documentation>The alignment option value byte-alignment indicates that the event codes and associated content are aligned on byte boundaries. While byte-alignment generally results in EXI streams of larger sizes compared with their bit-packed equivalents, byte-alignment may provide a help in some use cases that involve frequent copying of large arrays of scalar data directly out of the stream. It can also make it possible to work with data in-place and can make it easier to debug encoded data by allowing items on aligned boundaries to be easily located in the stream.</xs:documentation>
      </xs:annotation>
    </xs:enumeration>
    <xs:enumeration value='pre-compression'>
      <xs:annotation>
        <xs:documentation>The alignment option value pre-compression indicates that all steps involved in compression (see section 9. EXI Compression) are to be done with the exception of the final step of applying the DEFLATE algorithm. The primary use case of pre-compression is to avoid a duplicate compression step when compression capability is built into the transport protocol. In this case, pre-compression just prepares the stream for later compression.</xs:documentation>
      </xs:annotation>
    </xs:enumeration>
  </xs:restriction>
</xs:simpleType>

<xs:element name='uploadSchema'>
  <xs:complexType>
    <xs:simpleContent>
    </xs:simpleContent>
  </xs:complexType>
</xs:element>
<xs:extension base='xs:base64Binary'>
  <xs:attribute name='contentType' type='ContentType' use='optional' default='Text'/>
  <xs:attribute name='bytes' type='xs:positiveInteger' use='optional'/>
  <xs:attribute name='md5Hash' type='MD5Hash' use='optional'/>
</xs:extension>
</xs:simpleContent>
</xs:complexType>
</xs:element>

<xs:simpleType name='ContentType'>
<xs:restriction base='xs:string'>
  <xs:enumeration value='Text'/>
  <xs:enumeration value='ExiBody'/>
  <xs:enumeration value='ExiDocument'/>
</xs:restriction>
</xs:simpleType>

<xs:element name='downloadSchema' type='DownloadSchema'/>
<xs:element name='downloadSchemaResponse' type='DownloadSchemaResponse'/>

<xs:complexType name='DownloadSchema'>
  <xs:attribute name='url' type='xs:string' use='required'/>
</xs:complexType>

<xs:complexType name='DownloadSchemaResponse'>
  <xs:complexContent>
    <xs:extension base='DownloadSchema'>
      <xs:choice minOccurs='0' maxOccurs='1'>
        <xs:element name='httpError'>
          <xs:complexType>
            <xs:attribute name='code' type='xs:positiveInteger' use='required'/>
            <xs:attribute name='message' type='xs:string' use='required'/>
          </xs:complexType>
        </xs:element>
        <xs:element name='invalidUrl'>
          <xs:complexType>
            <xs:attribute name='message' type='xs:string' use='required'/>
          </xs:complexType>
        </xs:element>
        <xs:element name='timeout'>
          <xs:complexType>
            <xs:attribute name='message' type='xs:string' use='required'/>
          </xs:complexType>
        </xs:element>
      </xs:choice>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>
<xs:attribute name='message' type='xs:string' use='required'/>
</xs:complexType>
</xs:element>
<xs:element name='invalidContentType'>
<xs:complexType>
  <xs:attribute name='contentTypeReturned' type='xs:string' use='required'/>
</xs:complexType>
</xs:element>
<xs:element name='error'>
<xs:complexType>
  <xs:attribute name='message' type='xs:string' use='required'/>
</xs:complexType>
</xs:element>
</xs:choice>
<xs:attribute name='result' type='xs:boolean' use='required'/>
</xs:extension>
</xs:complexContent>
</xs:complexType>
<xs:element name='streamStart'>
<xs:complexType>
  <xs:sequence>
    <xs:element name='xmlns' minOccurs='0' maxOccurs='unbounded'>
      <xs:complexType>
        <xs:attribute name='prefix' type='xs:string' use='required'/>
        <xs:attribute name='namespace' type='xs:string' use='required'/>
      </xs:complexType>
    </xs:element>
    <xs:element ref='streams:features' minOccurs='0' maxOccurs='1'/>
    <xs:any namespace='urn:ietf:params:xml:ns:xmpp-tls' minOccurs='0' maxOccurs='1'/>
    <xs:any namespace='urn:ietf:params:xml:ns:xmpp-sasl' minOccurs='0' maxOccurs='1'/>
    <xs:element ref='streams:error' minOccurs='0' maxOccurs='1'/>
  </xs:sequence>
  <xs:attribute name='from' type='xs:string' use='optional'/>
  <xs:attribute name='id' type='xs:string' use='optional'/>
  <xs:attribute name='to' type='xs:string' use='optional'/>
</xs:complexType>
</xs:element>
8 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/esosntqewhs/iot-xmpp/.

9 Acknowledgements

Thanks to Joachim Lindborg, Takuki Kamiya, Tina Beckman, Karin Forsell, Jeff Freund and Rumen Kyusakov for all valuable feedback.