XEP-0332: HTTP over XMPP transport

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

2017-09-11
Version 0.5

Status	Type	Short Name
Deferred	Standards Track	NOT_YET_ASSIGNED

This specification defines how XMPP can be used to transport HTTP communication over peer-to-peer networks.

Legal

Copyright

This XMPP Extension Protocol is copyright © 1999 – 2018 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

Many documents have been written on how to transport XMPP datagrams using HTTP. The motivation behind such solutions has often been to be able to use XMPP in scripting languages such as Java Script running in web browsers.
But up to this point very little has been written about the reverse: How to transport HTTP methods and HTTP responses over an XMPP-based peer-to-peer network. Here, the motivation is as follows: There are multitudes of applications and APIs written that are based on HTTP over TCP as the basic communication transport protocol. As these are moving closer and closer to the users, problems arise when the users want to protect their data and services using firewalls. Even though there are methods today to open up firewalls manually or automatically permit communication with such devices and applications, you still open up the application for everybody. This rises the need for more advanced security measures which is sometimes difficult to implement using HTTP.
The XMPP protocol however does not have the same problems as HTTP in these regards. It's a peer-to-peer protocol naturally allowing communication with applications and devices behind firewalls. It also includes advanced user authentication and authorization which makes it easier to make sure unauthorized access to private content is prevented.
Furthermore, with the advent of semantic web technologies and its use in web 3.0 and Internet of Things applications, such applications move even more rapidly into the private spheres of the users, where security and privacy is of paramount importance, it is necessary to use more secure transport protocols than HTTP over TCP.
There are many different types of HTTP-based communication that one would like to be able to transport over XMPP. A non-exhaustive list can include:

- Web Content like pages, images, files, etc.
- Web Forms.
- Web Services (SOAP, REST, etc.)
- Semantic Web Resources (RDF, Turtle, etc.)
- Federated SPARQL queries (SQL-type query language for the semantic web, or web 3.0)
- Streamed multi-media content in UPnP and DLNA networks.

Instead of trying to figure out all possible things transportable over HTTP and make them transportable over XMPP, this document ignores the type of content transported, and instead focuses on encoding and decoding the original HTTP requests and responses, building an HTTP tunnel over an existing XMPP connection. It would enable existing applications to work seamlessly over XMPP if browsers and web services supported this extension (like displaying your home control application on your phone when you are at work), without the need to update the myriad of existing applications. It would also permit federated SPARQL queries in personal networks with the added benefit of being able to control who can talk to who (or

what can talk to what) through established friendship relationships.
Previous extensions handling different aspects of XMPP working together with HTTP:

- Verifying HTTP Requests via XMPP (XEP-0070) [1]: This specification handles client authentication of resources, where there are three parties: HTTP Client <-> HTTP Server/XMPP Client <-> XMPP Server. Here HTTP Client authentication to resources on the HTTP Server is made by a third party, an XMPP Server.
- SOAP over XMPP (XEP-0072) [2]: This specification handles execution of SOAP-based web services specifically. This specification has some benefits regarding to Web Service calls over XMPP, but is only one example of all types of HTTP-based communication one would desire to transport over XMPP.
- BOSH (XEP-0124) [3]: This specification handles XMPP-based communication over HTTP sessions (BOSH), allowing for instance, XMPP communication in java script using the XML HTTP Request object. This is in some way the reverse of what this document proposes to do.
- Stanza Headers and Internet Metadata (XEP-0131) [4]: While not directly related to HTTP, it is used to transport headers in the form of collections of key-value pairs, exactly as is done in HTTP. The format for encoding headers into XMP defined by this XEP will be re-used in this XEP.
- XMPP URI Query Components (XEP-0147) [5]: This informational specification proposes ways to define XMPP-actions using URL's. The xmpp URI scheme is formally defined in RFC 5122 [6]. This document will propose a different URI scheme for HTTP-based resources over an XMPP transport: httpx.

2 Requirements

This document presupposes the server already has a web server (HTTP Server) implementation, and that it hosts content through it, content which can be both dynamic (i.e. generated) or static (e.g. files) in nature. Content, which it wants to publish to XMPP clients as well as HTTP clients. It also presupposes that the client is aware of HTTP semantics and MIME encoding.

[1] XEP-0070: Verifying HTTP Requests via XMPP <https://xmpp.org/extensions/xep-0070.html>.
[2] XEP-0072: SOAP over XMPP <https://xmpp.org/extensions/xep-0072.html>.
[3] XEP-0124: Bidirectional-streams Over Synchronous HTTP <https://xmpp.org/extensions/xep-0124.html>.
[4] XEP-0131: Stanza Headers and Internet Metadata <https://xmpp.org/extensions/xep-0131.html>.
[5] XEP-0147: XMPP URI Query Components <https://xmpp.org/extensions/xep-0147.html>.
[6] RFC 5122: Internationalized Resource Identifiers (IRIs) and Uniform Resource Identifiers (URIs) for the Extensible Messaging and Presence Protocol (XMPP) <http://tools.ietf.org/html/rfc5122>.

3 Glossary

The following table lists common terms and corresponding descriptions.

HTTP Hyper Text Transfer Protocol. Version 1.1 of HTTP is described in RFC 2616 RFC 2616: Hypertext Transfer Protocol -- HTTP/1.1 <http://tools.ietf.org/html/rfc2616> . The PATCH method is described in RFC 5789 RFC 5789: PATCH Method for HTTP <http://tools.ietf.org/html/rfc5789>

HTTP Client An HTTP Client is the initiator of an HTTP Request.

HTTP Method HTTP Methods are: OPTIONS, GET, HEAD, POST, PUT, DELETE, TRACE and PATCH. The HTTP Method CONNECT is not supported by this specification.

HTTP Request An HTTP Request consists of a HTTP Method, version information, headers and optional body.

HTTP Resource A resource on an HTTP Server identified by a path. Each path begins with a separator character (/).

HTTP Response An HTTP Response consists of a status code, optional status message, headers and optional body.

HTTP Server An HTTP Server responds to HTTP Client requests.

Web Server Used synonymously with HTTP Server.

4 Use Cases

All HTTP communication is done using the **Request/Response** paradigm. Each HTTP Request is made sending an **iq**-stanza containing a **req** element to the server. Each **iq**-stanza sent is of type **set**.
When the server responds, it does so by sending an **iq**-stanza response (type **result**) back to the client containing a **resp** element. Since responses are asynchronous, and since multiple requests may be active at the same time, responses may be returned in a different order than the in which the original requests were made.
Requests or responses containing data must also consider how this data should be encoded within the XML telegram. Normally in HTTP, content and headers are separated by a blank line, and the transfer of the content is made in the same stream. Specific HTTP headers are used to define how the content is transferred and encoded within the stream (Content-Type, Content-Length, Content-Encoding, Content-Transfer-Encoding). This approach is not possible if the response is to be embedded in an XML telegram, since it can interfere with the encoding of the encompassing XML.
To solve this, this document specifies additional data transfer mechanisms that are compatible with the XMPP protocol. The normal HTTP-based content transfer headers will still

be transported, but do not affect the content encoding used in the XMPP transport. The following content encoding methods are available:

text Normal text content. The text is encoded as text within XML, using the same encoding used by the XML stream. XML escape characters (<, > and &) are escaped using the normal <, > and & character escape sequences.

xml Xml content embedded in the XML telegram. Note however, that any processing instructions or XML version statements must be avoided, since it may cause the XML stream to become invalid XML. If this is a problem, normal text encoding can be used as an alternative. The advantage of xml instead of text or base64 encodings is when used in conjunction with EXI compression Efficient XML Interchange (EXI) Format (XEP-0322) XEP-0322: Efficient XML Interchange (EXI) Format <https://xmpp.org/extensions/xep-0322.html>.. EXI compression has the ability to compress XML efficiently. Text will not be compressed, unless response exists in internal string tables. Base-64 encoded data will be compressed so that the 33% size gain induced by the encoding is recaptured.

base64 Base-64 encoded binary content. Can be used to easily embed binary content in the telegram.

chunkedBase64 Chunked Base-64 encoded binary content. The content is not embedded in the telegram. Instead it is sent in chunks, using separate chunk messages to the client. Chunked transport can be used by the server when it doesn't know the size of the final result. Streaming content, i.e. content of infinite length, must use ibb or jingle transport types to transfer content. If the content consists of a file, sipub should be used.

Chunked encoding is perfect for dynamic responses of moderate sizes, for instance for API method responses. The server does not know when the response is begun to be generated what the final size will be, but it will be most probably "manageable". Using the chunked transfer mechanism enables the server to start sending the content, minimizing the need for buffers, and at the same time minimizing the number of messages that needs to be sent, increasing throughput.

The client can limit the maximum chunk size to be used by the server, using the maxChunkSize attribute in the request. The chunk size can be set to a value between 256 and 65536. If not provided in the request, the server chooses an appropriate value. Note that chunks can be sent containing a smaller amount of bytes than the maximum chunk size provided in the request.

sipub The sender might deem the content to be too large for sending embedded in the XMPP telegram. To circumnavigate this, the sender publishes the content as a file using Publishing Stream Initiation Requests (XEP-0137) XEP-0137: Publishing Stream Initiation Requests <https://xmpp.org/extensions/xep-0137.html>. (Publishing Stream Initiation Requests), instead of embedding the content directly. This might be the case for instance, when a client requests a video resource, without using a ranged request.

This transfer mechanism is of course the logical choice, if the content is already stored in a file on the server, and the size of the file is sufficiently large to merit the overhead of sipub. Smaller files can simply be returned using the text, xml or base64 mechanisms.

The client can disable the use of sipub by the server, by including a sipub='false' attribute in the request. sipub is enabled by default. On constrained devices with limited support for different XEP's, this can be a way to avoid the use of technologies not supported by the client.

ibb This option may be used to encode indefinite streams, like live audio or video streams (HLS, SHOUTcast, Motion JPeg web cams, etc). It uses In-Band Bytestreams (XEP-0047) XEP-0047: In-Band Bytestreams <https://xmpp.org/extensions/xep-0047.html>. to send the content over an in-band bytestream to the client. This option is not available in requests, only in responses.

Streams must not use any of the above mechanisms. Only ibb and jingle mechanisms can be used. If the content represents multimedia jingle is preferrable, especially if different encodings are available.

The client can disable the use of ibb by the server, by including a ibb='false' attribute in the request. ibb is enabled by default. On constrained devices with limited support for different XEP's, this can be a way to avoid the use of technologies not supported by the client.

jingle For demanding multi-media streams alternative methods to transport streaming rather than embedded into the XMPP stream may be required. Even though the ibb method may be sufficient to stream a low-resolution web cam in the home, or listen to a microphone or a radio station, it is probably badly suited for high-resolution video streams with multiple video angles and audio channels. If such content is accessed and streamed, the server can negotiate a different way to stream the content using Jingle (XEP-0166) XEP-0166: Jingle <https://xmpp.org/extensions/xep-0166.html>..

The client can disable the use of ingle by the server, by including a jingle='false' attribute in the request. jingle is enabled by default. On constrained devices with limited support for different XEP's, this can be a way to avoid the use of technologies not supported by the client.

Note: Content encoded using **chunkedBase64** encoding method can be terminated, either by the receptor going off-line, or by sending a **close** command to the sender. The transfer methods **sipub**, **ibb** and **jingle** have their own mechanisms for aborting content transfer.

4.1 HTTP Methods

The following use cases show how different HTTP methods may work when transported over XMPP. To facilitate the readability in these examples, simple text or xml results are shown.

4.1.1 OPTIONS

This section shows an example of an OPTIONS method call. OPTIONS is described in §9.2 in RFC 2616.

Listing 1: OPTIONS

```
<iq type='set'
       from='httpclient@example.org/browser'
       to='httpserver@example.org'
       id='1'>
      <req xmlns='urn:xmpp:http' method='OPTIONS' resource='*' version
         ='1.1'>
          <headers xmlns='http://jabber.org/protocol/shim'>
              <header name='Host'>example.org</header>
          </headers>
      </req>
   </iq>

   <iq type='result'
       from='httpserver@example.org'
       to='httpclient@example.org/browser'
       id='1'>
      <resp xmlns='urn:xmpp:http' version='1.1' statusCode='200'
         statusMessage='OK'>
          <headers xmlns='http://jabber.org/protocol/shim'>
              <header name='Date'>Fri, 03 May 2013 13:52:10 GMT-4</
                 header>
              <header name='Allow'>OPTIONS, GET, HEAD, POST, PUT,
                 DELETE, TRACE</header>
              <header name='Content-Length'>0</header>
          </headers>
      </resp>
   </iq>
```

4.1.2 GET

This section shows an example of a GET method call. GET is described in §9.3 in RFC 2616.

Listing 2: GET

```
<iq type='set'
       from='httpclient@example.org/browser'
       to='httpserver@example.org'
       id='2'>
      <req xmlns='urn:xmpp:http' method='GET' resource='/rdf/xep'
         version='1.1'>
          <headers xmlns='http://jabber.org/protocol/shim'>
```

```
            <header name='Host'>example.org</header>
         </headers>
      </req>
   </iq>

   <iq type='result'
       from='httpserver@example.org'
       to='httpclient@example.org/browser'
       id='2'>
      <resp xmlns='urn:xmpp:http' version='1.1' statusCode='200'
         statusMessage='OK'>
          <headers xmlns='http://jabber.org/protocol/shim'>
              <header name='Date'>Fri, 03 May 2013 16:39:54GMT-4</
                 header>
              <header name='Server'>Clayster</header>
              <header name='Content-Type'>text/turtle</header>
              <header name='Content-Length'>...</header>
              <header name='Connection'>Close</header>
          </headers>
          <data>
              <text>@prefix dc: &lt;http://purl.org/dc/elements/1.1/&
                 gt;.
@base &lt;http://example.org/&gt;.

&lt;xep&gt; dc:title "HTTP over XMPP";
      dc:creator &lt;PeterWaher&gt;;
      dc:publisher &lt;XSF&gt;.</text>
          </data>
      </resp>
   </iq>
```

Note: The XMPP/HTTP bridge at the server only transmits headers literally as they are reported, as if it was normal HTTP over TCP that was used. In the HTTP over XMPP case, connections are not handled in the same way, and so the "Connection: Close" header has no meaning in this case. For more information about connection handling in the HTTP over XMPP case, see the section on Connection Handling.

4.1.3 HEAD

This section shows an example of a HEAD method call. HEAD is described in §9.4 in RFC 2616.

Listing 3: HEAD

```
<iq type='set'
       from='httpclient@example.org/browser'
       to='httpserver@example.org'
       id='3'>
```

```
      <req xmlns='urn:xmpp:http' method='HEAD' resource='/video/video1
          .m4' version='1.1'>
          <headers xmlns='http://jabber.org/protocol/shim'>
              <header name='Host'>example.org</header>
          </headers>
      </req>
   </iq>

   <iq type='result'
       from='httpserver@example.org'
       to='httpclient@example.org/browser'
       id='3'>
      <resp xmlns='urn:xmpp:http' version='1.1' statusCode='200'
          statusMessage='OK'>
          <headers xmlns='http://jabber.org/protocol/shim'>
              <header name='Date'>Fri, 03 May 2013 16:57:12GMT-4</
                  header>
              <header name='Server'>Clayster</header>
              <header name='Content-Type'>video/mp4</header>
              <header name='Content-Length'>12345678</header>
          </headers>
      </resp>
   </iq>
```

4.1.4 POST

This section shows an example of a POST method call. POST is described in §9.5 in RFC 2616.

Listing 4: POST

```
<iq type='set'
       from='httpclient@example.org/browser'
       to='httpserver@example.org'
       id='4'>
      <req xmlns='urn:xmpp:http' method='POST' resource='/sparql/?
          default-graph-uri=http%3A%2F%2Fexample.org%2Frdf/xep'
          version='1.1'>
          <headers xmlns='http://jabber.org/protocol/shim'>
              <header name='Host'>example.org</header>
              <header name='User-agent'>Clayster HTTP/XMPP Client</
                  header>
              <header name='Content-Type'>application/sparql-query</
                  header>
              <header name='Content-Length'>...</header>
          </headers>
          <data>
              <text>PREFIX dc: &lt;http://purl.org/dc/elements/1.1/&gt
                  ;
```

```
BASE &lt;http://example.org/&gt;

SELECT ?title ?creator ?publisher
WHERE  { ?x dc:title ?title .
         OPTIONAL { ?x dc:creator ?creator } .
       }</text>
          </data>
      </req>
   </iq>

   <iq type='result'
       from='httpserver@example.org'
       to='httpclient@example.org/browser'
       id='4'>
      <resp xmlns='urn:xmpp:http' version='1.1' statusCode='200'
         statusMessage='OK'>
          <headers xmlns='http://jabber.org/protocol/shim'>
              <header name='Date'>Fri, 03 May 2013 17:09:34-4</header>
              <header name='Server'>Clayster</header>
              <header name='Content-Type'>application/sparql-results+
                  xml</header>
              <header name='Content-Length'>...</header>
          </headers>
          <data>
              <xml>
                  <sparql xmlns="http://www.w3.org/2005/sparql-results
                     #">
                      <head>
                          <variable name="title"/>
                          <variable name="creator"/>
                      </head>
                      <results>
                          <result>
                              <binding name="title">
                                  <literal>HTTP over XMPP</literal>
                              </binding>
                              <binding name="creator">
                                  <uri>http://example.org/PeterWaher</
                                     uri>
                              </binding>
                          </result>
                      </results>
                  </sparql>
              </xml>
          </data>
      </resp>
   </iq>
```

Note: If using **xml** encoding of data, care has to be taken to avoid including the version and encoding information (<?xml version="1.0"?>) at the top of the document, otherwise the resulting XML will be invalid. Care has also to be taken to make sure that the generated XML is not invalid XMPP, even though it might be valid XML. This could happen for instance, if the XML contains illegal elements from the jabber:client namespace. If in doubt, use another encoding mechanism.

4.1.5 PUT

This section shows an example of a PUT method call. PUT is described in §9.6 in RFC 2616.

Listing 5: PUT

```
<iq type='set'
       from='httpclient@example.org/browser'
       to='httpserver@example.org'
       id='5'>
      <req xmlns='urn:xmpp:http' method='PUT' resource='/index.html'
         version='1.1'>
          <headers xmlns='http://jabber.org/protocol/shim'>
              <header name='Host'>example.org</header>
              <header name='Content-Type'>text/html</header>
              <header name='Content-Length'>...</header>
          </headers>
          <data>
              <text>&lt;html&gt;&lt;header/&gt;&lt;body&gt;&lt;p&gt;
                 Beautiful home page.&lt;/p&gt;&lt;/body&gt;&lt;/html
                 &gt;</text>
          </data>
      </req>
   </iq>

   <iq type='result'
       from='httpserver@example.org'
       to='httpclient@example.org/browser'
       id='5'>
      <resp xmlns='urn:xmpp:http' version='1.1' statusCode='204'
         statusMessage='No Content'>
          <headers xmlns='http://jabber.org/protocol/shim'>
              <header name='Date'>Fri, 03 May 2013 17:40:41GMT-4</
                 header>
              <header name='Content-Length'>0</header>
          </headers>
      </resp>
   </iq>
```

4.1.6 DELETE

This section shows an example of a DELETE method call. DELETE is described in §9.7 in RFC 2616.

Listing 6: DELETE

```
<iq type='set'
       from='httpclient@example.org/browser'
       to='httpserver@example.org'
       id='6'>
      <req xmlns='urn:xmpp:http' method='DELETE' resource='/index.html
         ' version='1.1'>
          <headers xmlns='http://jabber.org/protocol/shim'>
              <header name='Host'>example.org</header>
          </headers>
      </req>
   </iq>

   <iq type='result'
       from='httpserver@example.org'
       to='httpclient@example.org/browser'
       id='6'>
      <resp xmlns='urn:xmpp:http' version='1.1' statusCode='403'
         statusMessage='Forbidden'>
          <headers xmlns='http://jabber.org/protocol/shim'>
              <header name='Date'>Fri, 03 May 2013 17:46:07GMT-4</
                 header>
              <header name='Content-Type'>text/plain</header>
              <header name='Content-Length'>...</header>
          </headers>
          <data>
              <text>You're not allowed to change the home page!</text>
          </data>
      </resp>
   </iq>
```

4.1.7 TRACE

This section shows an example of a TRACE method call. TRACE is described in §9.8 in RFC 2616.

Listing 7: TRACE

```
<iq type='set'
       from='httpclient@example.org/browser'
       to='httpserver@example.org'
       id='7'>
```

```
      <req xmlns='urn:xmpp:http' method='TRACE' resource='/rdf/ex1.
         turtle' version='1.1'>
          <headers xmlns='http://jabber.org/protocol/shim'>
              <header name='Host'>example.org</header>
          </headers>
      </req>
   </iq>

   <iq type='result'
       from='httpserver@example.org'
       to='httpclient@example.org/browser'
       id='7'>
      <resp xmlns='urn:xmpp:http' version='1.1' statusCode='200'
         statusMessage='OK'>
          <headers xmlns='http://jabber.org/protocol/shim'>
              <header name='Date'>Fri, 03 May 2013 17:55:10GMT-4</
                 header>
              <header name='Server'>Clayster</header>
              <header name='Content-Type'>message/http</header>
              <header name='Content-Length'>...</header>
          </headers>
          <data>
              <text>GET /rdf/ex1.turtle HTTP/1.1
Host: example.org</text>
          </data>
      </resp>
   </iq>
```

Note: The Trace command returns the request it received from the client by the server. Here, however, it is assumed that the request is made over HTTP/TCP, not HTTP/XMPP. Therefore, in this example, the XMPP layer has transformed the HTTP/XMPP request into an HTTP/TCP-looking request, which is returned as the response to the TRACE Method call. RFC 2616 is silent to the actual format of the TRACE response (MIME TYPE message/http), and TRACE is only used (if not disabled for security reasons) for debugging connections and routing via proxies. Therefore, a response returning the original XMPP request should also be accepted by the caller.

4.1.8 PATCH

This section shows an example of a PATCH method call. PATCH is described in RFC 5789.

Listing 8: PATCH

```
<iq type='set'
       from='httpclient@example.org/browser'
       to='httpserver@example.org'
       id='8'>
```

```
    <req xmlns='urn:xmpp:http' method='PATCH' resource='/file.txt'
        version='1.1'>
        <headers xmlns='http://jabber.org/protocol/shim'>
            <header name='Host'>www.example.com</header>
            <header name='Content-Type'>application/example</header>
            <header name='If-Match'>e0023aa4e</header>
            <header name='Content-Length'>100</header>
        </headers>
        <data>
            [description of changes]
        </data>
    </req>
</iq>

<iq type='result'
    from='httpserver@example.org'
    to='httpclient@example.org/browser'
    id='8'>
    <resp xmlns='urn:xmpp:http' version='1.1' statusCode='204'
        statusMessage='No Content'>
        <headers xmlns='http://jabber.org/protocol/shim'>
            <header name='Content-Location'>/file.txt</header>
            <header name='ETag'>e0023aa4e</header>
        </headers>
    </resp>
</iq>
```

4.2 Encoding formats

In the following sub-sections, the different data encoding formats are discussed, each with corresponding examples to illustrate how they work. The interesting part of these examples is the **data** element and its contents.

4.2.1 text

Text responses is a simple way to return text responses (i.e. any MIME Type starting with text/). Since the text is embedded into XML, the characters <, > and & need to be escaped to <, > and & respectively.
The following example shows how a TURTLE response, which is text-based, is returned using the **text** encoding:

Listing 9: text

```
<iq type='result'
    from='httpserver@example.org'
    to='httpclient@example.org/browser'
```

```
       id='2'>
      <resp xmlns='urn:xmpp:http' version='1.1' statusCode='200'
         statusMessage='OK'>
          <headers xmlns='http://jabber.org/protocol/shim'>
              <header name='Date'>Fri, 03 May 2013 16:39:54GMT-4</
                 header>
              <header name='Server'>Clayster</header>
              <header name='Content-Type'>text/turtle</header>
              <header name='Content-Length'>...</header>
              <header name='Connection'>Close</header>
          </headers>
          <data>
              <text>@prefix dc: &lt;http://purl.org/dc/elements/1.1/&
                 gt;.
@base &lt;http://example.org/&gt;.

&lt;xep&gt; dc:title "HTTP over XMPP";
      dc:creator &lt;PeterWaher&gt;;
      dc:publisher &lt;XSF&gt;.</text>
          </data>
      </resp>
   </iq>
```

4.2.2 xml

XML is a convenient way to return XML embedded in the XMPP response. This can be suitable for MIME Types of the form **.*/(.*[+])?xml** (using regular expression to match them), like text/xml, application/soap+xml or application/sparql-results+xml. Care has to be taken however, since not all XML constructs can be embedded as content to an XML element without invalidating it, like the xml version and encoding declaration (<?xml version="1.0"?> as an example). Care has also to be taken to make sure that the generated XML is not invalid XMPP, even though it might be valid XML. This could happen for instance, if the XML contains illegal elements from the jabber:client namespace.
If unsure how to handle XML responses using the **xml** encoding type, you can equally well use the **text** type, but encode the XML escape characters <, > and &, or use another encoding, like **base64**.
The advantage of **xml** instead of **text** or **base64** encodings is when used in conjunction with EXI compression. EXI compression has the ability to compress XML efficiently. Text will not be compressed, unless response exists in internal string tables. Base-64 encoded data will be compressed so that the 33% size gain induced by the encoding is recaptured.

Listing 10: xml

```
<iq type='result'
       from='httpserver@example.org'
       to='httpclient@example.org/browser'
```

```
      id='4'>
    <resp xmlns='urn:xmpp:http' version='1.1' statusCode='200'
       statusMessage='OK'>
        <headers xmlns='http://jabber.org/protocol/shim'>
            <header name='Date'>Fri, 03 May 2013 17:09:34-4</header>
            <header name='Server'>Clayster</header>
            <header name='Content-Type'>application/sparql-results+
               xml</header>
            <header name='Content-Length'>...</header>
        </headers>
        <data>
            <xml>
                <sparql xmlns="http://www.w3.org/2005/sparql-results
                   #">
                    <head>
                        <variable name="title"/>
                        <variable name="creator"/>
                    </head>
                    <results>
                        <result>
                            <binding name="title">
                                <literal>HTTP over XMPP</literal>
                            </binding>
                            <binding name="creator">
                                <uri>http://example.org/PeterWaher</
                                   uri>
                            </binding>
                        </result>
                    </results>
                </sparql>
            </xml>
        </data>
    </resp>
</iq>
```

4.2.3 base64

Base-64 encoding is a simple way to encode content that is easily embedded into XML. Apart from the advantage of being easy to encode, it has the disadvantage to increase the size of the content by 33%, since it requires 4 bytes to encode 3 bytes of data. Care has to be taken not to send too large items using this encoding.

Note: The actual size of the content being sent does not necessarily need to increase if this encoding method is used. If EXI compression is used at the same time, and it uses schema-aware compression, it will actually understand that the character set used to encode the data only uses 6 bits of information per character, and thus compresses the data back to its original size.

The following example shows an image is returned using the **base64** encoding:

Listing 11: base64

```
<iq type='result'
       from='httpserver@example.org'
       to='httpclient@example.org/browser'
       id='9'>
      <resp xmlns='urn:xmpp:http' version='1.1' statusCode='200'
         statusMessage='OK'>
          <headers xmlns='http://jabber.org/protocol/shim'>
              <header name='Date'>Fri, 03 May 2013 16:39:54GMT-4</
                 header>
              <header name='Server'>Clayster</header>
              <header name='Content-Type'>image/png</header>
              <header name='Content-Length'>221203</header>
          </headers>
          <data>
              <base64>
                  iVBORw0KGgoAAAANSUhEUgAAASwAAAGQCAYAAAAUdV17AAAAAXNSR0
                   ... tVWJd+e+y1AAAAABJRU5ErkJggg==</base64>
          </data>
      </resp>
   </iq>
```

4.2.4 chunkedBase64

In HTTP, Chunked Transfer Encoding is used when the sender does not know the size of the content being sent, and to avoid having its buffers overflow, sends the content in chunks with a definite size.
A similar method exists in the HTTP over XMPP transport: The **chunkedBase64** allows the sender to transmit the content in chunks. Every chunk is base-64 encoded. The stream of chunks are identified by a **streamId** parameter, since chunks from different responses may be transmitted at the same time.
Another difference between normal chunked transport, and the **chunkedBase64** encoding, is that the size of chunks does not have to be predetermined. Chunks are naturally delimited and embedded in the XML stanza. The last chunk in a response must have the **last** attribute set to true.

Listing 12: chunkedBase64

```
<iq type='result'
       from='httpserver@example.org'
       to='httpclient@example.org/browser'
       id='10'>
      <resp xmlns='urn:xmpp:http' version='1.1' statusCode='200'
         statusMessage='OK'>
          <headers xmlns='http://jabber.org/protocol/shim'>
```

```
            <header name='Date'>Fri, 04 May 2013 13:43:12GMT-4</
               header>
            <header name='Server'>Clayster</header>
            <header name='Content-Type'>image/png</header>
            <header name='Content-Length'>221203</header>
        </headers>
        <data>
            <chunkedBase64 streamId='Stream0001'/>
        </data>
    </resp>
</iq>

<message from='httpserver@example.org'
         to='httpclient@example.org/browser'>
    <chunk xmlns='urn:xmpp:http' streamId='Stream0001' nr='0'>
       iVBORw0KGgoAAAANSUhEUgAAASwAAAGQCAYAA ...</chunk>
</message>

...

<message from='httpserver@example.org'
         to='httpclient@example.org/browser'>
    <chunk xmlns='urn:xmpp:http' streamId='Stream0001' nr='5' last='
       true'>... 2uPzi9u+tVWJd+e+y1AAAAABJRU5ErkJggg==</chunk>
</message>
```

Note: Chunked encoding assumes the content to be finite. If content is infinite (i.e. for instance live streaming), the **ibb** or **jingle** transfer encodings must be used instead. If the sender is unsure if the content is finite or infinite, **ibb** or **jingle** must be used.
Note 2: If the web server sends chunked data to the client it uses the HTTP header **Transfer-Encoding: chunked**, and then sends the data in chunks but with chunk sizes inserted so the receiving end can decode the incoming data. Note that this data will be included in the data sent in the XMPP chunks defined by this document. In this case, data will be chunked twice: First by the web server, and then by the HTTP over XMPP transport layer. When received by the client, it is first reassembled by the HTTP over XMPP layer on the client, and then by the HTTP client who will read the original chunk size elements inserted into the content. More information about HTTP chunking, can be found in RDF2616 §3.6.1.
Note 3: In order to work over XMPP servers that do not maintain message order, a **nr** attribute is available on the **chunk** element. The first chunk reports a **nr** of zero. Each succcessive chunk reports a **nr** that is incremented by one. In this way, the receiver can make sure to order incoming chunks in the correct order.

4.2.5 sipub

Often content being sent can be represented by a file, virtual or real, especially if the content actually represents a file and is not dynamically generated. In these instances, instead of

embedding the contents in the response, since content can be potentially huge, a File Stream Initiation is returned instead, as defined in XEP 0137: Publishing Stream Initiation Requests. This is done using the **sipub** element.

Listing 13: sipub

```
<iq type='result'
       from='httpserver@example.org'
       to='httpclient@example.org/browser'
       id='11'>
      <resp xmlns='urn:xmpp:http' version='1.1' statusCode='200'
         statusMessage='OK'>
          <headers xmlns='http://jabber.org/protocol/shim'>
              <header name='Date'>Fri, 03 May 2013 16:39:54GMT-4</
                 header>
              <header name='Server'>Clayster</header>
              <header name='Content-Type'>image/png</header>
              <header name='Content-Length'>221203</header>
          </headers>
          <data>
              <sipub xmlns='http://jabber.org/protocol/sipub'
                     from='httpserver@example.org'
                     id='file-0001'
                     mime-type='image/png'
                     profile='http://jabber.org/protocol/si/profile/
                        file-transfer'>
                  <file xmlns='http://jabber.org/protocol/si/profile/
                     file-transfer'
                        name='Kermit.png'
                        size='221203'
                        date='2013-03-06T16:47Z'/>
              </sipub>
          </data>
      </resp>
   </iq>
```

4.2.6 ibb

Some web servers provide streaming content, i.e. content where packets are sent according to a timely fashion. Examples are video and audio streams like HLS (HTTP Live Streams), SHOUTcast, ICEcast, Motion JPeg, etc. In all these examples, content is infinite, and cannot be sent "all as quickly as possible". Instead, content is sent according to some kind of bitrate or frame rate for example.
Such content must use the **ibb** transfer mechanism, if used (or the **jingle** transfer machanism). The **ibb** transfer mechanism uses In-Band Bytestreams to transfer data from the server to the client. It starts by sending an a **ibb** element containing a **sid** attribute identifying the stream. Then the server sends an **ibb:open** IQ-stanza to the client according to XEP-0047. The client

can choose to reject, negotiate or acceopt the request whereby the transfer is begun. When the client is satisified and wants to close the stream, it does so, also according to XEP-0047. The **sid** value returned in the HTTP response is the same **sid** value that is later used by the IBB messages that follow. In this way, the client can relate the HTTP request and response, with the corresponding data transferred separately.

Listing 14: ibb

```
<iq type='set'
       from='httpclient@example.org/browser'
       to='httpserver@example.org'
       id='12'>
      <req xmlns='urn:xmpp:http' method='GET' resource='/webcam1.jpg'
         version='1.1'>
          <headers xmlns='http://jabber.org/protocol/shim'>
              <header name='Host'>example.org</header>
          </headers>
      </req>
   </iq>

   <iq type='result'
       from='httpserver@example.org'
       to='httpclient@example.org/browser'
       id='12'>
      <resp xmlns='urn:xmpp:http' version='1.1' statusCode='200'
         statusMessage='OK'>
          <headers xmlns='http://jabber.org/protocol/shim'>
              <header name='Date'>Fri, 04 May 2013 15:05:32GMT-4</
                 header>
              <header name='Server'>Clayster</header>
              <header name='Content-Type'>multipart/x-mixed-replace;
                 boundary=__234792749283748923749283 7</header>
          </headers>
          <data>
              <ibb sid='Stream0002'/>
          </data>
      </resp>
   </iq>

   <iq type='set'
       from='httpserver@example.org'
       to='httpclient@example.org/browser'
       id='13'>
      <open xmlns='http://jabber.org/protocol/ibb'
            block-size='32768'
            sid='Stream0002'
            stanza='message'/>
   </iq>
```

```
<iq type='result'
    from='httpclient@example.org/browser'
    to='httpserver@example.org'
    id='13'/>

<message from='httpserver@example.org'
         to='httpclient@example.org/browser'>
   <data xmlns='http://jabber.org/protocol/ibb' sid='Stream0002'
      seq='0'>...</chunk>
</message>

...

<iq type='set'
    from='httpclient@example.org/browser'
    to='httpserver@example.org'
    id='14'>
   <close xmlns='http://jabber.org/protocol/ibb' sid='Stream0002'/>
</iq>

<iq type='result'
    from='httpserver@example.org'
    to='httpclient@example.org/browser'
    id='14'/>
```

4.2.7 jingle

For demanding multi-media streams alternative methods to transport streaming rather than embedded into the XMPP stream may be required. Even though the **ibb** method may be sufficient to stream a low-resolution web cam in the home, or listen to a microphone or a radio station, it is probably badly suited for high-resolution video streams with multiple video angles and audio channels. If such content is accessed and streamed, the server can negotiate a different way to stream the content using XEP 0166: Jingle.

Listing 15: jingle

```
<iq type='result'
      from='httpserver@example.org'
      to='httpclient@example.org/browser'
      id='14'>
     <resp xmlns='urn:xmpp:http' version='1.1' statusCode='200'
        statusMessage='OK'>
         <headers xmlns='http://jabber.org/protocol/shim'>
             <header name='Date'>Fri, 03 May 2013 16:39:54GMT-4</
                header>
             <header name='Server'>Clayster</header>
             <header name='Content-Type'>video/mp4</header>
             <header name='Content-Length'>...</header>
```

```
        </headers>
        <data>
            <jingle xmlns='urn:xmpp:jingle:1'
                    action='session-initiate'
                    initiator='romeo@montague.lit/orchard'
                    sid='a73sjjvkla37jfea'>
                <content creator='initiator' name='voice'>
                    <description xmlns='urn:xmpp:jingle:apps:rtp:1'
                        media='audio'>
                        <payload-type id='96' name='speex' clockrate
                            ='16000'/>
                        <payload-type id='97' name='speex' clockrate
                            ='8000'/>
                        <payload-type id='18' name='G729'/>
                        <payload-type id='0' name='PCMU' />
                        <payload-type id='103' name='L16' clockrate=
                            '16000' channels='2'/>
                        <payload-type id='98' name='x-ISAC'
                            clockrate='8000'/>
                    </description>
                    <transport xmlns='urn:xmpp:jingle:transports:ice
                        -udp:1'
                               pwd='asd88fgpdd777uzjYhagZg'
                               ufrag='8hhy'>
                        <candidate component='1'
                                   foundation='1'
                                   generation='0'
                                   id='el0747fg11'
                                   ip='10.0.1.1'
                                   network='1'
                                   port='8998'
                                   priority='2130706431'
                                   protocol='udp'
                                   type='host'/>
                        <candidate component='1'
                                   foundation='2'
                                   generation='0'
                                   id='y3s2b30v3r'
                                   ip='192.0.2.3'
                                   network='1'
                                   port='45664'
                                   priority='1694498815'
                                   protocol='udp'
                                   rel-addr='10.0.1.1'
                                   rel-port='8998'
                                   type='srflx'/>
                    </transport>
                </content>
            </jingle>
```

```
        </data>
      </resp>
    </iq>
```

Note: Example taken from XEP 166: Jingle.
Note2: Using Jingle in this way makes it possible for an intelligent server to return multiple streams the client can choose from, something that is not done in normal HTTP over TCP. The first candidate should however correspond to the same stream that would have been returned if the request had been made using normal HTTP over TCP.

4.3 Applications

The following section lists use cases based on type of application. It is used to illustrate what types of applications would benefit from implementing this extension.

4.3.1 Browsers

HTTP began as a protocol for presenting text in browsers. So, browsers is a natural place to start to list use cases for this extensions. In general, content is identified using URL's, and in the browser a user enters the URL into Address Field of the browser, and the corresponding content is displayed in the display area. The content itself will probably contain links to other content, each such item identified by an absolute or relative URL.
The syntax and format of Uniform Resource Locators (URLs) or Uniform Resource Identifiers (URIs) is defined in RFC 3986 [7]. The basic format is defined as follows:

Listing 16: URL syntax

```
URI = scheme ":" hier-part [ "?" query ] [ "#" fragment ]

   hier-part = "//" authority path-abempty
               / path-absolute
               / path-rootless
               / path-empty
```

RFC 2616 [8] furthermore defines the format of URLs using the http URI scheme, as follows:

Listing 17: HTTP (over TCP) URL syntax

```
http_URL = "http:" "//" host [ ":" port ] [ abs_path [ "?" query ]]
```

RFC 2818 [9] continues to define the https scheme for HTTP transport over SSL/TLS over TCP using the same format as for HTTP URLs, except the https scheme is used to inform the client

[7] RFC 3986: Uniform Resource Identifiers (URI): Generic Syntax <http://tools.ietf.org/html/rfc3986>.
[8] RFC 2616: Hypertext Transport Protocol -- HTTP/1.1 <http://tools.ietf.org/html/rfc2616>.
[9] RFC 2818: HTTP Over TLS <http://tools.ietf.org/html/rfc2818>.

that HTTP over SSL/TLS is to be used.
In a similar way, this document proposes a new URI scheme: **httpx**, based on the HTTP URL scheme, except **httpx** URLs imply the use of HTTP over XMPP instead of HTTP over TCP. URLs using the **httpx** URL scheme has the following format:

Listing 18: HTTP over XMPP URL syntax

```
httpx_URL = "httpx:" "//" resourceless_jid [ abs_path [ "?" query ]]
```

Here, the host and port parts of normal HTTP URLs have been replaced by the resource-less JID of the HTTP Server, i.e. only the user name, the @ character and the domain. The / separator between the resource-less JID and the following abs_path, is part of abs_path.

Listing 19: Examples of URLs with the httpx scheme

```
httpx://httpServer@example.org/index.html
   httpx://httpServer@example.org/images/image1.png
   httpx://httpServer@example.org/api?p1=a&p2=b
```

By creating a new scheme for HTTP over XMPP transport, and implementing support for it in web browsers, XML HTTP request objects and web servers, Web Applications previously requiring web hosting on the Internet will be able to be seamlessly hosted privately and securely behind firewalls instead, by simply switching from the http URL scheme to the httpx URL scheme in the calling application. All relative URL's within the application, including URL's sent to the XHR object (Ajax) will automatically be directed to use the HTTP over XMPP transport instead.
It's beyond the scope of this specification to define how browsers handles its own XMPP account(s) and roster. This section only makes a suggestion to show how this can be handled. It is assumed in this discussion that the browser has a working XMPP connection with a server, and has its own JID. For simplicity, we will assume the browser has only one connection. Extension to multiple connection is canonical.
When resolving an URL using the httpx scheme, the browser needs to extract the JID of the server hosting the resource. If that JID is already in the roster, the request can proceed as usual.
If not in the roster, the browser needs to send a friendship request. A non-exhaustive list of states could be made:

- No response: This could be presented as a connection to the content server being made.
- Request rejected: This could be handled in the same way as HTTP Error Forbidden.
- Request accepted: Connection made, proceed with fetching content.
- Timeout: If no friendship request response have been returned, the browser can choose to time out.

Since XMPP works both ways, the browser can receive friendship requests from the outside world. Any such requests should be displayed to the end user, if any, or rejected.
For more information, see Roster Handling in web clients and Roster Handling in web servers.
Today, most people who want to host their own web applications (HTML/HTTP based applications) need to host them on a server publicly available on the Internet. However, many applications of a private nature like a family blog, home automation system, etc., is not suited for public hosting, since it puts all private data at risk of being compromised, or access to home security functions (like home web cams) to get in the hands of people you don't want to have access to them.
To solve this, one can host the application on a server at home, perhaps a small cheap plug computer consuming as little as 1 or 2 Watts of electricity, using a web server supporting this extension. If the following design rules are followed, the application should be visible in any browser also supporting this extensions, as long as friendship exists between the browser and the web server:

- Only relative URL's are used within references (images, audio, video, links, objects, etc.). If absolute URL's are used (including scheme), the browser might get the first page correctly, but will be unable to get the content with the absolute URL, unless the URL has the same scheme as the principal page.

- URL's to web forms must also be relative, for the same reason.

- Any URL's sent to the XML HTTP Request (XHR) Object directed to API's or resources hosted by the same application must also be relative, for the same reasons as above. The XHR Object supports relative URL's.

If the above rules are met, which they should under normal conditions, typing in the httpx URL in the browser (for instance when you're at the office) should display the application (hosted for example at home behind a firewall) in the same way as when you use http (or https) when you have access to the server (for instance when you're home), as long as friendship exists between the browser JID and the server JID.

4.3.2 Web Services

Many applications use a Service Oriented Architecture (SOA) and use web services to communicate between clients and servers. These web services are mostly HTTP over TCP based, even though there are bindings which are not based on this. The most common APIs today (REST) are however all based on HTTP over TCP. Being HTTP over TCP requires the web server hosting the web services either to be public or directly accessible by the client. But as the services move closer to end users (for instance a Thermostat publishing a REST API for control in your home), problems arise when you try to access the web service outside of

private network in which the API is available. As explained previously, the use of HTTP over XMPP solves this.
The following example shows a simple SOAP method call:

Listing 20: SOAP method call

```
<iq type='set'
       from='httpclient@example.org/browser'
       to='httpserver@example.com'
       id='15'>
      <req xmlns='urn:xmpp:http' method='POST' resource='/Math'
         version='1.1'>
          <headers xmlns='http://jabber.org/protocol/shim'>
              <header name='Host'>www.example.com</header>
              <header name='Content-Type'>application/soap+xml;
                  charset=utf-8</header>
              <header name='Content-Length'>...</header>
          </headers>
          <data>
              <xml>
                  <soap:Envelope xmlns:soap="http://www.w3.org
                     /2001/12/soap-envelope"
                                 soap:encodingStyle="http://www.w3.org
                                    /2001/12/soap-encoding">
                      <soap:Body xmlns:m="http://www.example.org/math"
                         >
                          <m:AddNumbers>
                              <m:N1>10</m:N1>
                              <m:N2>20</m:N2>
                          </m:GetStockPrice>
                      </soap:Body>
                  </soap:Envelope>
              </xml>
          </data>
      </req>
   </iq>

   <iq type='result'
       from='httpserver@example.com'
       to='httpclient@example.org/browser'
       id='15'>
      <resp xmlns='urn:xmpp:http' version='1.1' statusCode='200'
         statusMessage='OK'>
          <headers xmlns='http://jabber.org/protocol/shim'>
              <header name='Content-Type'>application/soap+xml;
                  charset=utf-8</header>
              <header name='Content-Length'>...</header>
          </headers>
          <data>
```

```
            <xml>
                <soap:Envelope xmlns:soap="http://www.w3.org
                    /2001/12/soap-envelope"
                                soap:encodingStyle="http://www.w3.org
                                    /2001/12/soap-encoding">
                    <soap:Body xmlns:m="http://www.example.org/math"
                        >
                        <m:AddNumbersResponse>
                            <m:Sum>30</m:Sum>
                        </m:AddNumbersResponse>
                    </soap:Body>
                </soap:Envelope>
            </xml>
        </data>
    </resp>
</iq>
```

Note: Other components of SOAP, such as WSDL and disco-documents are just examples of content handled by simple GET requests.
This section shows an example of a REST method call. REST method calls are just simple GET, POST, PUT or DELETE HTTP calls with dynamically generated content.

Listing 21: REST

```
<iq type='set'
      from='httpclient@example.org/browser'
      to='httpserver@example.org'
      id='16'>
    <req xmlns='urn:xmpp:http' method='GET' resource='/api/
        multiplicationtable?m=5' version='1.1'>
        <headers xmlns='http://jabber.org/protocol/shim'>
            <header name='Host'>example.org</header>
        </headers>
    </req>
</iq>

<iq type='result'
    from='httpserver@example.org'
    to='httpclient@example.org/browser'
    id='16'>
    <resp xmlns='urn:xmpp:http' version='1.1' statusCode='200'
        statusMessage='OK'>
        <headers xmlns='http://jabber.org/protocol/shim'>
            <header name='Date'>Fri, 05 May 2013 15:01:53GMT-4</
                header>
            <header name='Server'>Clayster</header>
            <header name='Content-Type'>text/xml</header>
            <header name='Content-Length'>...</header>
        </headers>
```

```
        <data>
            <xml>
                <table>
                    <value n='1' nTimesM='5'/>
                    <value n='2' nTimesM='10'/>
                    <value n='3' nTimesM='15'/>
                    <value n='4' nTimesM='20'/>
                    <value n='5' nTimesM='25'/>
                    <value n='6' nTimesM='30'/>
                    <value n='7' nTimesM='35'/>
                    <value n='8' nTimesM='40'/>
                    <value n='9' nTimesM='45'/>
                    <value n='10' nTimesM='50'/>
                </table>
            </xml>
        </data>
    </resp>
</iq>
```

4.3.3 Semantic Web & IoT

The Semantic Web was originally developed as a way to link data between servers on the Web, and understand it. However, with the advents of technologies such as SPARQL, the Semantic Web has become a way to unify API's into a universal form of distributed API to all types of data possible. It also allows for a standardized way to perform grid computing, in the sense that queries can be federated and executed in a distributed fashion ("in the grid").
For these reasons, and others, semantic web technologies have been moving closer to Internet of Things, and also into the private spheres of its end users. Since the semantic web technologies are based on HTTP, they also suffer from the shortcomings of HTTP over TCP, when it comes to firewalls and user authentication and authorization. Allowing HTTP transport over XMPP greatly improves the reach of semantic technologies beyond "The Internet" while at the same time improving security and controllability of the information.
As the semantic web moves closer to Internet of Things and the world of XMPP, it can benefit from work done with relation to the Internet of Things, such as Internet of Things - Provisioning (XEP-0324) [10], which would give automatic control of who (or what) can communicate with whom (or what).
Turtle [11], is a simple way to represent semantic data. The following example shows Turtle-encoded semantic data being returned to the client as a response to a request.

Listing 22: Turtle

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

[10] XEP-0324: Internet of Things - Provisioning <https://xmpp.org/extensions/xep-0324.html>.
[11] Turtle: Terse RDF Triple Language <http://www.w3.org/TR/turtle/>

```
        to='httpclient@example.org/browser'
        id='2'>
       <resp xmlns='urn:xmpp:http' version='1.1' statusCode='200'
          statusMessage='OK'>
           <headers xmlns='http://jabber.org/protocol/shim'>
               <header name='Date'>Fri, 03 May 2013 16:39:54GMT-4</
                  header>
               <header name='Server'>Clayster</header>
               <header name='Content-Type'>text/turtle</header>
               <header name='Content-Length'>...</header>
               <header name='Connection'>Close</header>
           </headers>
           <data>
               <text>@prefix dc: &lt;http://purl.org/dc/elements/1.1/&
                  gt;.
@base &lt;http://example.org/&gt;.

&lt;xep&gt; dc:title "HTTP over XMPP";
       dc:creator &lt;PeterWaher&gt;;
       dc:publisher &lt;XSF&gt;.</text>
           </data>
       </resp>
   </iq>
```

RDF [12], is a another way to represent semantic data, better suited than Turtle for M2M communication. Related technologies, such as the micro format RDFa [13] allows for embedding RDF into HTML pages or XML documents. The following example shows RDF-encoded semantic data being returned to the client as a response to a request.

Listing 23: RDF

```
<iq type='result'
       from='httpserver@example.org'
       to='httpclient@example.org/browser'
       id='17'>
       <resp xmlns='urn:xmpp:http' version='1.1' statusCode='200'
          statusMessage='OK'>
           <headers xmlns='http://jabber.org/protocol/shim'>
               <header name='Date'>Fri, 05 May 2013 16:02:23GMT-4</
                  header>
               <header name='Server'>Clayster</header>
               <header name='Content-Type'>application/rdf+xml</header>
               <header name='Content-Length'>...</header>
           </headers>
           <data>
               <xml>
```

[12] RDF: Resource Description Framework <http://www.w3.org/RDF/>
[13] RDFa: RDF through attributes <http://www.w3.org/TR/rdfa-syntax/>

```
                    <rdf:RDF xmlns:dc="http://purl.org/dc/elements/1.1/"
                             xmlns:rdf="http://www.w3.org/1999/02/22-rdf
                                -syntax-ns#">
                        <rdf:Description rdf:about="http://example.org/
                            xep">
                            <dc:title>HTTP over XMPP</dc:title>
                            <dc:creator rdf:resource="http://example.org
                                /PeterWaher" />
                            <dc:publisher rdf:resource="http://example.
                                org/XSF" />
                        </rdf:Description>
                    </rdf:RDF>
                </xml>
            </data>
        </resp>
    </iq>
```

This section shows an example of a SPARQL query executed as a POST call.

Listing 24: SPARQL

```
<iq type='set'
       from='httpclient@example.org/browser'
       to='httpserver@example.org'
       id='4'>
      <req xmlns='urn:xmpp:http' method='POST' version='1.1'
           resource='/sparql/?default-graph-uri=http%3A%2F%2Fanother.
               example%2Fcalendar.rdf'>
          <headers xmlns='http://jabber.org/protocol/shim'>
              <header name='Host'>example.org</header>
              <header name='User-agent'>Clayster HTTP/XMPP Client</
                  header>
              <header name='Content-Type'>application/sparql-query</
                  header>
              <header name='Content-Length'>...</header>
          </headers>
          <data>
              <text>@prefix dc: &lt;http://purl.org/dc/elements/1.1/&
                  gt;.
@base &lt;http://example.org/&gt;.

&lt;xep&gt; dc:title "HTTP over XMPP";
      dc:creator &lt;PeterWaher&gt;;
      dc:publisher &lt;XSF&gt;.</text>
          </data>
      </req>
   </iq>

   <iq type='result'
```

```
      from='httpserver@example.org'
      to='httpclient@example.org/browser'
      id='4'>
    <resp xmlns='urn:xmpp:http' version='1.1' statusCode='200'
       statusMessage='OK'>
        <headers xmlns='http://jabber.org/protocol/shim'>
            <header name='Date'>Fri, 03 May 2013 17:09:34-4</header>
            <header name='Server'>Clayster</header>
            <header name='Content-Type'>application/sparql-results+
               xml</header>
            <header name='Content-Length'>...</header>
        </headers>
        <data>
            <xml>
                <sparql xmlns="http://www.w3.org/2005/sparql-results
                   #">
                    <head>
                        <variable name="title"/>
                        <variable name="creator"/>
                    </head>
                    <results>
                        <result>
                            <binding name="title">
                                <literal>HTTP over XMPP</literal>
                            </binding>
                            <binding name="creator">
                                <uri>http://example.org/PeterWaher</
                                   uri>
                            </binding>
                        </result>
                    </results>
                </sparql>
            </xml>
        </data>
    </resp>
</iq>
```

4.3.4 Streaming

There are many types of streams and streaming protocols. Several of these are based on HTTP or variants simulating HTTP. Examples of such HTTP-based or pseudo-HTTP based streaming protocols can include HLS [14] used for multi-media streaming, SHOUTcast [15] used for internet radio and Motion JPeg [16] common format for web cameras.
Common for all streaming data, is that they are indefinite, but at the same time rate-limited

[14] HLS: HTTP Live Streaming <http://en.wikipedia.org/wiki/HTTP_Live_Streaming>
[15] SHOUTcast <http://en.wikipedia.org/wiki/SHOUTcast>
[16] Motion JPeg <http://en.wikipedia.org/wiki/Motion_JPEG>

depending on quality, etc. Because of this, the web server is required to use the ibb encoding or the jingle encoding to transport the content to the client.

5 Determining Support

If an entity supports the protocol specified herein, it MUST advertise that fact by returning a feature of "urn:xmpp:http" in response to Service Discovery (XEP-0030) [17] information requests.

Listing 25: Service discovery information request

```
<iq type='set'
    from='httpclient@example.org/browser'
    to='httpserver@example.org'
    id='disco1'>
  <query xmlns='http://jabber.org/protocol/disco#info'/>
</iq>
```

Listing 26: Service discovery information response

```
<iq type='result'
    from='httpserver@example.org'
    to='httpclient@example.org/browser'
    id='disco1'>
  <query xmlns='http://jabber.org/protocol/disco#info'>
    ...
    <feature var='urn:xmpp:http'/>
    ...
  </query>
</iq>
```

In order for an application to determine whether an entity supports this protocol, where possible it SHOULD use the dynamic, presence-based profile of service discovery defined in Entity Capabilities (XEP-0115) [18]. However, if an application has not received entity capabilities information from an entity, it SHOULD use explicit service discovery instead.

6 Implementation Notes

6.1 Connection handling

HTTP over TCP includes headers for connection handling. The basic sequence for an HTTP request might be:

[17] XEP-0030: Service Discovery <https://xmpp.org/extensions/xep-0030.html>.
[18] XEP-0115: Entity Capabilities <https://xmpp.org/extensions/xep-0115.html>.

- Client connects to server
- Clients sends request
- Client received response
- Client closes connection

However, in the HTTP over XMPP case, there are no connections between the client and the server. Both clients and servers have active connections to the XMPP Server, but these remain unchanged during the sequence of requests. Therefore, both clients and servers should ignore any HTTP over TCP connection settings, since they have no meaning in the HTTP over XMPP case. However, the corresponding headers should always be transported as is, to maintain the information.

6.2 HTTP Headers

HTTP Headers are serialized to and from XML using the XEP-0131 Stanza Headers and Internet Metadata. However, this does not mean that the SHIM feature needs to be published by the client, as defined in §3 in XEP-0131, since the headers will be embedded into the HTTP elements. Also, if there is any conflicts in how to store header values, when it comes to data types, etc., the original format as used by the original HTTP request must be used, and not the format defined in Header Definitions or A Note About date-Related Headers in XEP-0131.
The HTTP over XMPP tunnel is just a tunnel of HTTP over XMPP, it does not know the semantic meaning of headers used in the transport. It does not know if additional headers added by the myriad of custom applications using HTTP are actually HTTP-compliant. It just acts as a transport, returning the sime kind of response (being deterministric) as if the original request was made through other means, for example over TCP. It does not add, remove or change semantic meaning of keys and values, nor change the format of the corresponding values. Such changes will create uncountable problems very difficult to detect and solve in a general case.
This specification differs from XEP-0131 in that this specification the headers are consumed by web servers and web clients (The XMPP client here only being a ”dumb” gateway), while in XEP-0131 the headers are consumed by the XMPP clients themselves, knowing XML and XML formats.

6.3 Stanza Sizes

Some XMPP Servers may limit stanza sizes for various reasons. While this may work well for certain applications, like Instant Messaging and Chat, implementors of HTTP over XMPP need to know that some server have such stanza size restrictions. Therefore, an implementation should include configurable size limits, so chunking can be used instead of sending large

stanzas. Another limit could be when streaming should be used instead of chunking. This later limit should be applied in particular on audio and video content.
The implementor should also consider to send large content in the form of files using file transfer, and large multi-media content using Jingle.
Note: According to RFC 6120 [19] there is a smallest allowed maximum stanza size that all XMPP servers must support. According to §13.12.4 of that document, this limit is set to 10000 bytes including all characters from the opening < character to the closing > character.

6.4 Bandwidth Limitations

Some XMPP Servers may also have bandwidth resitrctions enforced. This to limit the possibility of Denial of Service attacks or similar flooding of messages. Implementors of the HTTP over XMPP extensions must know however, that the bandwidth limitations for instant messaging and chat may be completely different from that of normal web applications. In chatting, a 1000 bytes/s limit is in most cases sufficient, while the same limit for even a modest web applications will make the application completely impossible to use.

7 Security Considerations

It's beyond the scope of this document to define how HTTP clients or HTTP servers handle rosters internally. The following sections list suggestions on how these can be handled by different parties.

7.1 Roster handling in browsers

Since browsers are operated by end users, any friendship request received from the outside should be either shown to the user (if the browser also maintains an IM client), or automatically rejected.
On the other hand, when the browser wants to access an URL using the httpx scheme, an automatic friendship request to the corresponding JID should be done, if not already in the roster. It is assumed that by entering the URL, or using the URL of an application already displayed, this implies giving permission to add that JID as a friend to the roster of the browser.

7.2 Roster handling in web servers

A web server should have different security settings available. The following subsections list possible settings for different scenarios. Note that these settings only reflect roster handling and cannot be set per resource. However, the server can maintain a set of JIDs with different

[19] RFC 6120: Extensible Messaging and Presence Protocol (XMPP): Core <http://tools.ietf.org/html/rfc6120>.

settings and restrict access to parts of the content hosted by the server per JID.

7.2.1 Public Server

A public server should accept requests from anybody (reachable from the current JID). All friendship requests should be automatically accepted.
To avoid bloating the roster, friendship requests could be automatically unsubscribed once the HTTP session has ended.

7.2.2 Manual Server

All new friendship are shown (or queued) to an administrator for manual acceptance or rejection. Once accepted, the client can access the corresponding content. During the wait (which can be substantial), the client should display a message that the friendship request is sent and response is pending.
Automatic unsubscription of friendships should only be done on a much longer inactivity timeframe than the normal session timeout interval.

7.2.3 Private Server

All new friendship requests are automatically rejected. Only already accepted friendships are allowed to make HTTP requests to the server.

7.2.4 Provisioned Server

All new friendship requests are delegated to a trusted third party, according to XEP 0324: Internet of Things - Provisioning. Friendship acceptance or rejection is then performed according to the response from the provisioning server(s).
Automatic friendship unsubscription can be made to avoid bloating the roster. However, the time interval for unsubscribing inactive users should be longer than the normal session timeout period, to avoid spamming any provisioning servers each time a client requests friendship.

8 IANA Considerations

The httpx URL scheme, as described above, must be registered as a provisional URI scheme according to BCP 35 [20]. The registration procedure is specified in BCP 35, section 7.

[20] BCP 35: New URI Schemes <http://tools.ietf.org/html/bcp35>

8.1 URI Scheme Registration Template

Following is an URI Scheme Registration Template, as per BCP 35, section 7.4.

URI scheme name httpx

Status provisional

URI scheme syntax The syntax used for the httpx scheme reuses the URI scheme syntax for the http scheme, as defined in RFC 2616:

http_URL = "http:" "//" host [":" port] [abs_path ["?" query]]

Instead of using host and port to define the where the HTTP server resides, the httpx scheme uses a resource-less XMPP JID to define where the HTTP server resides, implying the use of HTTP over XMPP as defined in this document instead of HTTP over TCP:

httpx_URL = "httpx:" "//" resourceless_jid [abs_path ["?" query]]

Here, the host and port parts of normal HTTP URLs have been replaced by the resource-less JID of the HTTP Server, i.e. only the user name, the @ character and the domain. The / separator between the resource-less JID and the following abs_path, is part of abs_path.

httpx://httpServer@example.org/index.html httpx://httpServer@example.org/images/image1.png httpx://httpServer@example.org/api?p1=a&p2=b

URI scheme semantics By creating a new scheme for HTTP over XMPP transport, and implementing support for it in web browsers, XML HTTP request objects and web servers, Web Applications previously requiring web hosting on the Internet will be able to be seamlessly hosted privately and securely behind firewalls instead, by simply switching from the http URL scheme to the httpx URL scheme in the calling application. All relative URL's within the application, including URL's sent to the XHR object (Ajax) will automatically be directed to use the HTTP over XMPP transport instead.

Encoding considerations Encoding is fully described in the Encoding section of the HTTP over XMPP XEP document.

Applications/protocols that use this URI scheme name This URI scheme is to be used in the same environments and by the same types of applications as the http URI scheme.

Interoperability considerations Interoperability considerations is described in the Implementation Notes section of the HTTP over XMPP XEP document.

Security considerations Security considerations is described in the Security considerations section of the HTTP over XMPP XEP document.

Contact For further information, please contact Peter Waher:

Email: peter.waher@example.org JabberID: peter.waher@jabber.org URI: http://www.linkedin.com/in/peterwaher

Author/Change controller For concerns regarding changes to the provisional registration, please contact Peter Waher:

Email: peter.waher@example.org JabberID: peter.waher@jabber.org URI: http://www.linkedin.com/in/peterwaher

References Following is a short list of referenced documents:

RFC 6120: Extensible Messaging and Presence Protocol (XMPP): Core

RFC 2616: Hypertext Transfer Protocol -- HTTP/1.1

HTTP over XMPP XEP: XMPP Extension Protocol: HTTP over XMPP

9 XMPP Registrar Considerations

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

10 XML Schema

```
<?xml version='1.0' encoding='UTF-8'?>
<xs:schema
    xmlns:xs='http://www.w3.org/2001/XMLSchema'
    targetNamespace='urn:xmpp:http'
    xmlns='urn:xmpp:http'
    xmlns:shim='http://jabber.org/protocol/shim'
    xmlns:sipub='http://jabber.org/protocol/sipub'
    xmlns:ibb='http://jabber.org/protocol/ibb'
    xmlns:jingle='urn:xmpp:jingle:1'
    elementFormDefault='qualified'>

    <xs:import namespace='http://jabber.org/protocol/shim'/>
    <xs:import namespace='http://jabber.org/protocol/sipub'/>
    <xs:import namespace='http://jabber.org/protocol/ibb'/>
    <xs:import namespace='urn:xmpp:jingle:1'/>

    <xs:element name='req'>
        <xs:complexType>
            <xs:sequence>
                <xs:element ref='shim:headers' minOccurs='0' maxOccurs
                    ='1'/>
                <xs:element name='data' type='Data' minOccurs='0'
                    maxOccurs='1'/>
            </xs:sequence>
            <xs:attribute name='method' type='Method' use='required'/>
            <xs:attribute name='resource' type='xs:string' use='
                required'/>
```

```
            <xs:attribute name='version' type='Version' use='required'
                />
            <xs:attribute name='maxChunkSize' type='MaxChunkSize' use=
                'optional'/>
            <xs:attribute name='sipub' type='xs:boolean' use='optional
                ' default='true'/>
            <xs:attribute name='ibb' type='xs:boolean' use='optional'
                default='true'/>
            <xs:attribute name='jingle' type='xs:boolean' use='
                optional' default='true'/>
        </xs:complexType>
    </xs:element>

    <xs:simpleType name='MaxChunkSize'>
        <xs:restriction base='xs:int'>
            <xs:minInclusive value='256'/>
            <xs:maxInclusive value='65536'/>
        </xs:restriction>
    </xs:simpleType>

    <xs:element name='resp'>
        <xs:complexType>
            <xs:sequence>
                <xs:element ref='shim:headers' minOccurs='0' maxOccurs
                    ='1'/>
                <xs:element name='data' type='Data' minOccurs='0'
                    maxOccurs='1'/>
            </xs:sequence>
            <xs:attribute name='version' type='Version' use='required'
                />
            <xs:attribute name='statusCode' type='xs:positiveInteger'
                use='required'/>
            <xs:attribute name='statusMessage' type='xs:string' use='
                optional'/>
        </xs:complexType>
    </xs:element>

    <xs:complexType name='Data'>
        <xs:choice minOccurs='1' maxOccurs='1'>
            <xs:element name='text' type='xs:string'>
                <xs:annotation>
                    <xs:documentation>Used for text responses that are
                        not XML.</xs:documentation>
                </xs:annotation>
            </xs:element>
            <xs:element name='xml'>
                <xs:annotation>
                    <xs:documentation>Specifically used for XML-
                        formatted responses.</xs:documentation>
```

```
            </xs:annotation>
            <xs:complexType>
                <xs:sequence minOccurs='0' maxOccurs='1'>
                    <xs:any processContents="lax"  namespace="##
                        any"/>
                </xs:sequence>
            </xs:complexType>
        </xs:element>
        <xs:element name="base64" type="xs:base64Binary">
            <xs:annotation>
                <xs:documentation>Short binary responses, base-64
                    encoded.</xs:documentation>
            </xs:annotation>
        </xs:element>
        <xs:element name="chunkedBase64">
            <xs:annotation>
                <xs:documentation>Content is divided into chunks
                    of binary base-64 encoded data.</
                    xs:documentation>
                <xs:documentation>Used if content is generated
                    dynamically and/or content size is not known.<
                    /xs:documentation>
                <xs:documentation>For streaming data the ibb:open
                    or jingle:jingle transports must be used.</
                    xs:documentation>
                <xs:documentation>For static data, such as files,
                    sipub:sipub should be used.</xs:documentation>
            </xs:annotation>
            <xs:complexType>
                <xs:attribute name="streamId" type="xs:string" use
                    ="required"/>
            </xs:complexType>
        </xs:element>
        <xs:element ref='sipub:sipub'>
            <xs:annotation>
                <xs:documentation>Content available through file
                    transfer.</xs:documentation>
            </xs:annotation>
        </xs:element>
        <xs:element name="ibb">
            <xs:annotation>
                <xs:documentation>Content returned through an in-
                    band bytestream.</xs:documentation>
            </xs:annotation>
            <xs:complexType>
                <xs:attribute name="sid" type="xs:string" use="
                    required"/>
            </xs:complexType>
        </xs:element>
```

```
            <xs:element ref="jingle:jingle">
                <xs:annotation>
                    <xs:documentation>Multi-media content returned
                        through jingle.</xs:documentation>
                </xs:annotation>
            </xs:element>
        </xs:choice>
    </xs:complexType>

    <xs:element name="chunk">
        <xs:complexType>
            <xs:simpleContent>
                <xs:extension base="xs:base64Binary">
                    <xs:attribute name="streamId" type="xs:string" use
                        ="required"/>
                    <xs:attribute name="nr" type="
                        xs:nonNegativeInteger" use="required"/>
                    <xs:attribute name="last" type="xs:boolean" use="
                        optional" default="false"/>
                </xs:extension>
            </xs:simpleContent>
        </xs:complexType>
    </xs:element>

    <xs:element name='close'>
        <xs:complexType>
            <xs:attribute name='streamId' type='xs:string' use='
                required'/>
        </xs:complexType>
    </xs:element>

    <xs:simpleType name='Method'>
        <xs:restriction base='xs:string'>
            <xs:enumeration value='OPTIONS'/>
            <xs:enumeration value='GET'/>
            <xs:enumeration value='HEAD'/>
            <xs:enumeration value='POST'/>
            <xs:enumeration value='PUT'/>
            <xs:enumeration value='DELETE'/>
            <xs:enumeration value='TRACE'/>
            <xs:enumeration value='PATCH'/>
        </xs:restriction>
    </xs:simpleType>

    <xs:simpleType name='Version'>
        <xs:restriction base='xs:string'>
            <xs:pattern value='\d[.]\d'/>
        </xs:restriction>
    </xs:simpleType>
```

```
</xs:schema>
```

11 Acknowledgements

Thanks to Peter Saint-Andre, Karin Forsell, Matthew A. Miller, Kevin Smith and Ralph Meijer for all valuable feedback.