XEP-0244: IO Data

Johannes Wagener
mailto:johannes.wagener@med.uni-muenchen.de
xmpp:edrin@jabber.org

Egon Willighagen
mailto:egonw@users.sf.net
xmpp:egonw@jabber.org

Andreas Heusler
mailto:aheusler@in.tum.de
xmpp:krach@jabber.org

Tobias Markmann
mailto:tm@ayena.de
xmpp:tm@ayena.de

Ola Spjuth
mailto:ola.spjuth@farmbio.uu.se
xmpp:olas@pele.farmbio.uu.se

2008-06-18
Version 0.1

Status	Type	Short Name
Deferred	Standards Track	NOT_YET_ASSIGNED

This specification defines an XMPP protocol extension for handling the input to and output from a remote entity.

Legal

Copyright

Permissions

Warranty

Liability

Conformance

Contents

1 Introduction

1.1 About the intention to write this XEP

Ad-Hoc Commands (XEP-0050) [1] became a popular and widespread XMPP Protocol Extension to execute functions on a remote systems. It is supported by many XMPP client and service implementations. To date almost all of its implementations rely on Data Forms (XEP-0004) [2] to be the data container. However Ad-Hoc Commands is explicitly designed and mentioned to be used in combination with other data containers, too. This applies for the cases where the Data Forms specification does not fit the needs, for example the Data Forms can be too restrictive on strong typing of data (see Section 1.2).
The intention of the present XEP is to define a data container for the cases where Data Forms is not applicable or not optimal. The data container defined herein (IO Data) is very generic and discoverable. It is intended to be used for other purposes than Data Forms.

1.2 Limitations of Data Forms

The Data Forms data container has certain restrictive limitations:

1. The supported Field Types are limited: text input fields, drop down boxes, different selectable optional values, etc. See Data Forms - Field Types
2. The only allowed content type of the fields is xs:string. See Data Forms - XML Schema
3. According to current specifications it is not possible to transport complex tree-based data structures. For example nested elements of elements cannot have nested elements at all, therefore lacking an XML key feature.

The limitations of Data Forms are not bad. They are good for the special use case a client has to render a graphical representation of the service. In HTML the correlative is a HTML form. For a chat client developer this makes it plain and simple to develop a generic graphical client implementation with some simple text-input fields.

1.3 Why an additional data container is required

According to current standards it is not supported to encapsulate more complex data in the Data Forms data container. For example it is not possible to encapsulate a complete XML Document - the real ”generic” data container - in the Data Forms data container, unless you encode the XML Document as xs:string – which would be considered bad practice.
However specialized clients are developed to make use of the service oriented architecture of XMPP. An example is given here: a XMPP client implementation reflects an Application

[1] XEP-0050: Ad-Hoc Commands <https://xmpp.org/extensions/xep-0050.html>.
[2] XEP-0004: Data Forms <https://xmpp.org/extensions/xep-0004.html>.

Programming Interface (API) with an XMPP services by making use of Ad-Hoc Commands.

Listing 1: get_titles.js

```
// javascript syntax

service = client.getService("database.server.com");

function = service.getFunction("getXmlFile");

// Check if this Ad-Hoc Command supports the IO Data XEP?
if (function instanceof IoDataFunction) {

   // Discover the input and output XML Schemata - required only once!
   schemata = function.getIoSchemata();

   // Dynamically marshal an API for the input and output - required
      only once!
   inputOutputFactory = new IoFactory( schemata );

   // Create and set the input
   input1 = inputOutputFactory.createInputObject();
   input1.setFileName("doc-ID-011021");

   // Invoke the function and get the output
   result1 = function.invoke(input1);
   output1 = inputOutputFactory.getOutputObject(result1);
   title1 = output1.getDocumentTitle();

   // Next document
   input2 = inputOutputFactory.createInputObject();
   input2.setFileName("doc-ID-833423");
   result2 = function.invoke(input2);
   output2 = inputOutputFactory.getOutputObject(result2);
   title2 = output1.getDocumentTitle();

   ...
}
```

The limitations of Data Forms make it impossible to define and handshake these actions clearly and precisely and without confusing existing and future implementations for the following reasons:

1. Data Forms does not support a "Schemata Discovery". The form descriptor Data Forms provides (type='form') is not separated from the data transaction according to the Ad-Hoc Command logic descibed in XEP-0050. Therefore each function invocation would result in a form descriptor submission again causing unnecessary traffic. It is sufficient to discover the IO Schemata once.

2. It is not suggested to encapsulate XML Documents in the Data Forms in general.

1.4 Web Services over XMPP

Beside Ad-Hoc Commands two other XEPs exist that provide mechanisms to execute a function on a remote system. For this count Jabber-RPC (XEP-0009) [3] and SOAP over XMPP (XEP-0072) [4].
However, Jabber-RPC and SOAP over XMPP lack certain functionality that is important for flexible, simple and robust Web Services. Because of the limited expressiveness of XML-RPCs data types the Jabber-RPC is not suitable for complex functionality, similar to the limitations of Data Forms. While SOAP over XMPP supports complex data types it lacks an obvious mechanism for asynchronous usage. For example it has no default stateful design: there is no sessionid like in Ad-Hoc Commands. Beside this SOAP brings in severe complexity (XML associated abstractions) that was required for the primary transport layer HTTP. This complexity is not required because XMPP does already implement the required XML associated abstractions. In addition to that there are other issues that argument against SOAP. For example to date most HTTP SOAP implemented services are only compatible with a subset of SOAP libraries.
In contrast Ad-Hoc Commands comprises simple, clean and optionally stateful Web Service mechanisms by default. In addition to that asynchronous client notification can be achieved with a <message>, as indicated in Ad-Hoc Commands and as realized in some unofficial implementations.

1.5 The IO Data data container

In conclusion and as already suggested in Ad-Hoc Commands we describe an alternative data container. This data container is more generic in the way it can be used:

1. It supports a ”Schemata Discovery”. Thus a client implementation can marshal an API for the input and output (and optionally for a service specific error) of a certain service.
2. This ”Schemata Discovery” is separated from the data transaction. This reduces the amount of unnecessary traffic.
3. The Field Types of the described data container are on the one hand clearly defined (there is only description, input, output, error, and status) and on the other hand straightforward. Thus any kind of XML data (XML Document with namespaces that represent any imaginable data object) can be submitted.

It is important to note that this XEP does not intent to replace or extent Data Forms. Also it does not break any current Ad-Hoc implementations. It just intends to offer another data

[3] XEP-0009: Jabber-RPC <https://xmpp.org/extensions/xep-0009.html>.
[4] XEP-0072: SOAP over XMPP <https://xmpp.org/extensions/xep-0072.html>.

container that fits much better under some circumstances where no GUI is rendered around an Ad-Hoc Command service.

2 Protocol

The base syntax for the 'urn:xmpp:tmp:io-data' namespace is as follows (see Protocol Namespaces regarding issuance of one or more permanent namespaces); a formal description can be found in the XML Schema section below.

```
<iodata xmlns='urn:xmpp:tmp:io-data'
      type='transaction-type'>
      <desc/>
      <in/>
      <out/>
      <error/>
      <status>
         <elapsed/>
         <remaining/>
         <percentage/>
         <information/>
      </status>
  </iodata>
```

2.1 Transaction Types

Transaction Type	Purpose	Associated Ad-Hoc Command	REQUIRED for generic XEP compatibility	Contained Elements
io-schemata-get	To request the schemata of input and output.	execute	yes	-
input	To submit the input.	execute	yes	<in>
getStatus	To request the status of the procedure.	next	yes	-
getOutput	To request the output.	next, complete	yes	-

Transaction Type	Purpose	Associated Ad-Hoc Command status value	REQUIRED for generic XEP compatibility	Contained Elements
io-schemata-result	To return the schemata of input and output.	completed	yes	<desc> <in> <out>
output	To submit the output.	executing, completed	yes	<out>
error	To submit additional error information.	executing	no	<error>
status	To indicate the current status of the procedure.	executing	no	<status>

2.2 Container Elements

<desc> -- a textual description of the IO Data data container (xs:string).
<in> -- contains the input. Valid for Transaction Type 'input' and 'io-schemata-result' only. May contain any XML data (XML Schema, XML Document ...).
<out> -- contains the output. Valid for Transaction Type 'output' and 'io-schemata-result' only. May contain any XML data (XML Schema, XML Document ...).
<error> -- describes the error raised by the procedure invocation. This element is optional and valid for Transaction Type 'error' and 'io-schemata-result' only. May contain any XML data (XML Schema, XML Document ...).
<status> -- describes the status of the procedure. This element is optional and valid for Transaction Type 'status' only.

2.3 Status Elements

<elapsed> -- an integer value of the time in milliseconds that elapsed since the procedure was invoked (xs:integer).
<remaining> -- an integer value of the (estimated) time in milliseconds till the procedure will finish (xs:integer).
<percentage> -- the percentage of the procedure that is finished (xs:integer).
<information> -- describes the current status of the procedure.

3 Implementation Notes

Commands (= remote procedures) executed with Ad-Hoc Commands and IO Data SHOULD NOT keep the requester in an uncertain state. This means the responder SHOULD respond to the requester always as fast as possible. Thereby the requester acquires the sessionid. (As some remote procedures/calculations are cost-intensive and/or time-consuming the requester MUST ”save” this sessionid for the case a network problem occurs.)
The Ad-Hoc Command logic applied for the IO Data data container should be associated with the following rules and keywords:

Ad-Hoc Command	Keyword	Associated Transaction Type	Subsequently allowed commands	Status description
execute	Get Schemata	io-schemata-get	-	XML Schemata are returned immediately
execute	Start procedure	input	-	output returns immediately (synchronous)
	Start procedure	input	next, cancel	asynchronous procedure was invoked
next	Check status	getStatus	next, cancel	asynchronous procedure not finished
	Check status	getStatus	next, complete, cancel	asynchronous procedure finished
	Get result	getOutput	next, complete, cancel	result was delivered
cancel	Cancel/delete procedure	-	-	procedure terminated
complete	Get result + delete procedure	-	-	result was delivered, procedure terminated

3.1 Synchronous implementation - for immediately completing services

1. If a service can return the output immediately, it MAY respond with status=’completed’ and return the output (IO Data type=’output’). This behavior is NOT RECOMMENDED for procedures that need more than 5 seconds to complete or that are cost-intensive.

3.2 Asynchronous implementation - for services that cannot return the output immediately

1. If a service cannot return the result immediately (this refers to procedures that need more than 5 seconds to complete) or the invoked procedure is cost-intensive, it SHOULD response with status='executing' and a <actions> element containing the <next> element.

2. If the service returned status='executing' the requester MAY stay up-to-date by proceeding with action='next' combined with the IO Data transaction type='getStatus'. The responder MUST respond with status='executing' and a <actions> element containing the <next> element only as long as the procedure is not finished.

3. If the procedure finished the responder MUST respond to this request (action='next') combined with the IO Data transaction type='getStatus' with status='executing' and a <actions> element containing the <next> and the <complete> elements to indicate that the output is ready for collection. The requester MAY then request the result by proceeding with action='complete' or action='next' combined with the IO Data transaction type='getOutput'.

4. Asynchronous notification: If the procedure finished the service MUST actively notify the requester by sending a message containing an Ad-Hoc Command element with status='executing' and a <actions> element containing the <next> and the <complete> elements to indicate that the result is ready for collection.

5. If the requester requests the output with action='complete' the responder MUST return the result (IO Data transaction type='output') with status='completed'. This means the Ad-Hoc Command session terminated. The responder MUST subsequently delete associated procedure and result.

6. If the requester requests the output with action='next' combined with the IO Data transaction type='getOutput' the responder MUST return the result (IO Data transaction type='output') with status='executing' and a <actions> element containing the <next> and the <complete> elements to indicate that the the Ad-Hoc Command session continues to exist and the output is still available. The requester MUST subsequently delete the associated procedure and result with action='cancel'.

3.3 Error handling

Beside the errors that are associated with IQ or Ad-Hoc Command abstraction layer an internal procedure error may occur.

1. If the procedure invocation fails (an error occurs) the responder MUST respond with status='completed'. To indicate that the procedure failed the <note> element MUST

have type='error' as described in XEP 50 Ad-Hoc Commands. The service may provide additional error information within the IO Data data container (IO Data transaction type='error').

2. Asynchronous implementation only: If the service returned status='executing' (asynchronous implementation) and the procedure fails (an error occurs) the service MUST actively notify the requester by sending a message containing an Ad-Hoc Command element with status='executing' and a <actions> element containing the <next> element to the invoker. To indicate that the procedure failed the <note> element MUST have type='error' as described in XEP 50 Ad-Hoc Commands. The service may provide additional error information within the IO Data data container (IO Data transaction type='error'). The requester SHOULD subsequently delete the associated procedure with action='cancel'.

3. Asynchronous implementation only: If the procedure failed (an error occurs) the responder MUST respond to a status request (action='next') with status='executing' and a <actions> element containing the <next> element to the requester. To indicate that the procedure failed the <note> element MUST have type='error' as described in XEP 50 Ad-Hoc Commands. The service may provide additional error information within the IO Data data container (IO Data transaction type='error'). The requester SHOULD subsequently delete the associated procedure with action='cancel'.

3.4 Optional status information

1. As long as the procedure did not finish (!) the service MAY provide additional status information within the IO Data data container (IO Data transaction type='status').

Formalising machine to machine commands using the namespace defined herein, making such commands detectable and usable on-the-fly without the prerequisite for the requester to know the exact interface on the service site and the support for asynchronous as well as synchronous execution contributes to the usability of XMPP for complex grid-computing projects.
In example an IDE could support the development of such projects by generating code interfaces (client stubs) to machine to machine capable XMPP services by discovering and requesting all required information on-the-fly.

4 Use Cases

4.1 Discovering support

The requester can query for disco information on the command (Ad-Hoc Command) node to find out if it supports IO Data based commands.

Listing 2: Disco request for command information

```
<iq type='get'
    from='user@university.example.org'
    to='service.university.example.org'
    id='iq_125'>
  <query xmlns='http://jabber.org/protocol/disco#info'
      node='get_threedimensionalcoordinates'/>
</iq>
```

Listing 3: Disco result for command information

```
<iq type='result'
    from='service.university.example.org'
    to='user@university.example.org'
    id='iq_125'>
  <query xmlns='http://jabber.org/protocol/disco#info'
      node='get_threedimensionalcoordinates'>
    <feature var='http://jabber.org/protocol/commands'/>
    <feature var='urn:xmpp:tmp:io-data'/>
  </query>
</iq>
```

To indicate support for IO Data it MUST include <feature var='urn:xmpp:tmp:io-data'/>. Of course the node can still provide <feature var='jabber:x:data'/> if this is supported, too.

4.2 Retrieving the IO Schemata

The 'in' and 'out' elements may each have any valid XML encoded elements as children. From a XML document style type of view <in/> and <out/> may be seen as root elements. Therefore it is required to "discover" the XML Schemata of the "dynamic children" of <in/> and <out/> (IO Schemata). This way a requester can marshal an API for the input and output of a certain service.
Beside the 'in' and 'out' elements an 'error' element is optionally allowed and would be discovered in exactly the same. It is not included in the example to keep it simple.
The XML Schemata request is done by setting the type of the IO Data element to 'io-schemata-get'.

Listing 4: IO Schemata request

```
<iq type='set'
    from='user@university.example.org'
    to='service.university.example.org'
    id='iq_126'>
  <command xmlns='http://jabber.org/protocol/commands'
      node='get_threedimensionalcoordinates'>
    <iodata xmlns='urn:xmpp:tmp:io-data' type='io-schemata-get'/>
```

```
  </command>
</iq>
```

Listing 5: IO Schemata result

```
<iq type='result'
    from='service.university.example.org'
    to='user@university.example.org'
    id='iq_126'>
  <command xmlns='http://jabber.org/protocol/commands'
      node='get_threedimensionalcoordinates'
      status='completed'>
    <iodata xmlns='urn:xmpp:tmp:io-data' type='io-schemata-result'>
      <desc>
        This service returns 3D atomic coordinates for the
        input structure. The input and output is encoded using the
        Chemical Markup Language (CML).
      </desc>
      <in>
        <xs:complexType>
          <xs:any namespace='http://www.xml-cml.org/schema'
              minOccur='1' maxOccur='1'/>
        </xs:complexType>
      </in>
      <out>
        <xs:complexType>
          <xs:any namespace='http://www.xml-cml.org/schema'
              minOccur='1' maxOccur='1'/>
        </xs:complexType>
      </out>
    </iodata>
  </command>
</iq>
```

This service example requires the content of <in/> and <out/> to be Chemical Markup Language [5] by requiring input with the namespace 'http://www.xml-cml.org/schema'. Additionally, it also defines the returned output to be Chemical Markup Language.

4.3 Executing a service

To keep the example simple the children of the 'in' and 'out' elements just contain strings (the protein name and protein sequence). However in real use cases it is likely that the children of 'in' and 'out' contain very complex XML documents with many different valid elements, namespaces, or values.
The requester transmits the input to the service (responder) by setting the type of the IO Data

[5] The Chemical Markup Language: <http://www.xml-cml.org/>.

element to 'input'.

Listing 6: Execute Command request

```
<iq type='set'
    from='user@university.example.org'
    to='service.university.example.org'
    id='iq_127'>
  <command xmlns='http://jabber.org/protocol/commands'
      node='get_proteinsequence'
      action='execute'>
    <iodata xmlns='urn:xmpp:tmp:io-data' type='input'>
      <in>
        <proteinname xmlns='http://university.example.org/protocol/
            proteinservice/proteinname'>
          CAB08284
        </proteinname>
      </in>
    </iodata>
  </command>
</iq>
```

The service transmits the output to the requester by setting the type of the IO Data element to 'output'.

Listing 7: Execute command result

```
<iq type='result'
    from='service.university.example.org'
    to='user@university.example.org'
    id='iq_127'>
  <command xmlns='http://jabber.org/protocol/commands'
      sessionid='RPC-SESSION-0000001'
      node='get_proteinsequence'
      status='completed'>
    <iodata xmlns='urn:xmpp:tmp:io-data' type='output'>
      <out>
        <proteinsequence xmlns='http://university.example.org/protocol
            /proteinservice/proteinsequence'>
          mrkhpqsatk hlfvsggvas slgkgltass lgqlltargl hvtmqkldpy
              lnvdpgtmnp
          fqhgevfvte dgaetdldvg hyerfldrdl sgsanvttgq vystviaker
              rgeylgdtvq
          viphitdeik qrimamaqpd ggdnrpdvvi teiggtvgdi esqpfleaar
              qvrhdlgren
          vfflhvslvp hlapsgelkt kptqhsvaal rsigitpdal ilrcdrdvpe
              slknkialmc
          dvdidgvist pdapsiydip kvlhreelda fvvrrlnlpf rdvdwtewdd
              llrrvhephg
```

```
            tvrialvgky vdfsdaylsv sealhaggfk hyakvevvwv asddcetatg
                aaavladvhg
            vlipggfgir giegkigair yararglpvl glclglqciv ieatrsvglv
                qansaefepa
            tpdpvistma dqkeivagea dfggtmrlga ypavlqpasi vaqaygttqv
                serhrhryev
            nnayrdwiae sglrisgtsp dgylvefvey panmhpfvvg tqahpelksr
                ptrphplfva
            fvgaaidyks aellpveipa vpeisehlpn ssnqhrdgve rsfpapaarg
        </proteinsequence>
      </out>
    </iodata>
  </command>
</iq>
```

4.4 Executing a time-consuming service

In this example the Ad-Hoc Command is a time-consuming and cost-intensive computation service. To keep the example simple the computation is a WAV to MP3 encoder - the input and output elements of this example make use of Bits of Binary (XEP-0231) [6].

Listing 8: Execute Command request

```
<iq type='set'
    from='user@server.org'
    to='encoder.server.org'
    id='iq_128'>
  <command xmlns='http://jabber.org/protocol/commands'
      node='wav2mp3'
      action='execute'>
    <iodata xmlns='urn:xmpp:tmp:io-data' type='input'>
      <in>
        <data xmlns='urn:xmpp:tmp:data-element'
            alt='my-song.wav'
            type='audio/x-wav'>
              [ ... base64-encoded-audio ... ]
        </data>
      </in>
    </iodata>
  </command>
</iq>
```

The service notifies the requester that the job is accepted: status='executing' and a <actions> element contains the <next> element.

[6] XEP-0231: Bits of Binary <https://xmpp.org/extensions/xep-0231.html>.

Listing 9: Execute command result

```
<iq type='result'
    from='encoder.user.org'
    to='user@user.org'
    id='iq_128'>
  <command xmlns='http://jabber.org/protocol/commands'
      sessionid='RPC-SESSION-0000002'
      node='wav2mp3'
      status='executing'>
    <note type='info'>WAV to MP3 encoding has been started. You may
       stay up to
        date using the next-action.
    </note>
    <actions>
      <next/>
    </actions>
  </command>
</iq>
```

The requester MAY stay up-to-date by proceeding with action='next' combined with the IO Data transaction type='getStatus'.

Listing 10: Execute Command request

```
<iq type='set'
    from='user@server.org'
    to='encoder.server.org'
    id='iq_129a'>
  <command xmlns='http://jabber.org/protocol/commands'
      sessionid='RPC-SESSION-0000002'
      node='wav2mp3'
      action='next'>
    <iodata xmlns='urn:xmpp:tmp:io-data' type='getStatus'>
  </command>
</iq>
```

The service returns the status of the procedure. The "still calculating"-status is indicated with the <actions> element that contains the <next> element only. The "calculation finished"-status is indicated with the <actions> element that contains the <next> and <complete> elements.
Optionally the result MAY contain additional status information within the IO Data element with IO Data transaction type='status' although is not shown here to keep the example simple.

Listing 11: Execute command result

```
<iq type='result'
    from='encoder.user.org'
    to='user@user.org'
```

```
    id='iq_129a'>
  <command xmlns='http://jabber.org/protocol/commands'
      sessionid='RPC-SESSION-0000002'
      node='wav2mp3'
      status='executing'>
    <actions>
       <next/>
    </actions>
  </command>
</iq>
```

If the procedure is complete the service notifies the invoker with a message stanza containing an Ad-Hoc Command namespace with status='executing' and a <actions> element that contains the <next> and <complete> elements. The <complete> element indicates the calculation finished.

Listing 12: Notification message

```
<message from='encoder.user.org'
    to='user@user.org'>
  <command xmlns='http://jabber.org/protocol/commands'
      sessionid='RPC-SESSION-0000002'
      node='wav2mp3'
      status='executing'>
    <note type='info'>WAV to MP3 encoding finished. You may request
        the
        output now.</note>
    <actions>
       <complete/>
       <next/>
    </actions>
  </command>
</iq>
```

After that the requester can request the output with the Ad-Hoc Command action='complete'.

Listing 13: Execute Command request

```
<iq type='set'
    from='user@server.org'
    to='encoder.server.org'
    id='iq_130a'>
  <command xmlns='http://jabber.org/protocol/commands'
      sessionid='RPC-SESSION-0000002'
      node='wav2mp3'
      action='complete'/>
</iq>
```

The service returns the MP3 within the IO Data element. The status of the Ad-Hoc Command completed (status='completed').

Listing 14: Execute command result

```
<iq type='result'
    from='encoder.user.org'
    to='user@user.org'
    id='iq_130a'>
  <command xmlns='http://jabber.org/protocol/commands'
      sessionid='RPC-SESSION-0000002'
      node='wav2mp3'
      status='completed'>
    <iodata xmlns='urn:xmpp:tmp:io-data' type='output'>
      <out>
        <data xmlns='urn:xmpp:tmp:data-element'
            alt='my-song.mp3'
            type='audio/mpeg'>
              [ ... base64-encoded-audio ... ]
        </data>
      </out>
    </iodata>
  </command>
</iq>
```

Alternatively the requester can request the output with the Ad-Hoc Command action='next' combined with the IO Data transaction type='getOutput'. This will keep the Ad-Hoc Command session alive and it must be deleted subsequently. This design allows to recover from network breakage during the result transmission state of the client-server communication, but allowing to request receiving the computation results or second time, because the session was left open after the first request.

Listing 15: Execute Command request

```
<iq type='set'
    from='user@server.org'
    to='encoder.server.org'
    id='iq_130b'>
  <command xmlns='http://jabber.org/protocol/commands'
      sessionid='RPC-SESSION-0000002'
      node='wav2mp3'
      action='next'>
    <iodata xmlns='urn:xmpp:tmp:io-data' type='getOutput'>
  </command>
</iq>
```

The service returns the MP3 within the IO Data element. The status of the Ad-Hoc Command remains active (status='executing').

Listing 16: Execute command result

```
<iq type='result'
    from='encoder.user.org'
    to='user@user.org'
    id='iq_130b'>
  <command xmlns='http://jabber.org/protocol/commands'
      sessionid='RPC-SESSION-0000002'
      node='wav2mp3'
      status='executing'>
    <actions>
       <complete/>
       <next/>
    </actions>
    <iodata xmlns='urn:xmpp:tmp:io-data' type='output'>
      <out>
        <data xmlns='urn:xmpp:tmp:data-element'
            alt='my-song.mp3'
            type='audio/mpeg'>
               [ ... base64-encoded-audio ... ]
        </data>
      </out>
    </iodata>
  </command>
</iq>
```

The requester MUST subsequently delete the remote procedure with the Ad-Hoc Command action='cancel'.

Listing 17: Execute Command request

```
<iq type='set'
    from='user@server.org'
    to='encoder.server.org'
    id='iq_131'>
  <command xmlns='http://jabber.org/protocol/commands'
      sessionid='RPC-SESSION-0000002'
      node='wav2mp3'
      action='cancel'/>
</iq>
```

The remote procedure is deleted.

Listing 18: Execute command result

```
<iq type='result'
    from='encoder.user.org'
    to='user@user.org'
    id='iq_130'>
  <command xmlns='http://jabber.org/protocol/commands'
```

```
      sessionid='RPC-SESSION-0000002'
      node='wav2mp3'
      status='canceled'/>
</iq>
```

4.5 Asynchronous error notification

In case of an error the service the service notifies the invoker with a message stanza containing an Ad-Hoc Command namespace with status='executing' and a <actions> element that contains the <next> element. In addition to that it MUST contain a <note> element with type='error' to indicate the error.
The error notification MAY contain additional error information within the IO Data element with IO Data transaction type='error'.

Listing 19: Error notification message

```
<message from='encoder.user.org'
    to='user@user.org'>
  <command xmlns='http://jabber.org/protocol/commands'
      sessionid='RPC-SESSION-0000002'
      node='wav2mp3'
      status='executing'>
    <note type='error'>#593 - The encoder could not parse the file.</
       note>
    <actions>
       <next/>
    </actions>
    <iodata xmlns='urn:xmpp:tmp:io-data' type='error'>
      <error>
        <errorcode/>593</errorcode>
        <description>The encoder could not parse the file.</
           description>
      </error>
    </iodata>
  </command>
</iq>
```

In case of an error the service would respond to a status request (Ad-Hoc Command action='next' combined with the IO Data transaction type='getStatus') in a very similar way except that a <iq> and not a <message> would be used.

4.6 Canceling a time-consuming service

An asynchronous remote procedure may be canceled (deleted) by the invoker at any time.

Listing 20: Cancel Command request

```
<iq type='set'
    from='user@server.org'
    to='encoder.server.org'
    id='iq_129b'>
  <command xmlns='http://jabber.org/protocol/commands'
      node='wav2mp3'
      action='cancel'/>
</iq>
```

The remote procedure is deleted.

Listing 21: Cancel command result

```
<iq type='result'
    from='encoder.user.org'
    to='user@user.org'
    id='iq_129b'>
  <command xmlns='http://jabber.org/protocol/commands'
      sessionid='RPC-SESSION-0000002'
      node='wav2mp3'
      status='canceled'/>
</iq>
```

5 Error Codes

Error codes on the Ad-Hoc Command abstraction layer are inherited from Ad-Hoc Commands. Application specific errors associated with a remote procedure call realized with IO Data in combination with Ad-Hoc Commands were described in section 3 - Implementation notes.

6 Internationalization Considerations

Internationalization of messages sent by the server is covered by setting the @xml:lang attribute of the <iq> element. Services should reply in the same language in which the client asked the question. That is, if the client specifies a locale using the @xml:lang attribute on the <iq> element, then the server should reply in the same locale, and localize messages given in <desc>, <node>@info and <query><item>@name.

7 Security Considerations

To follow.

8 IANA Considerations

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

9 XMPP Registrar Considerations

9.1 Protocol Namespaces

Until this specification advances to a status of Draft, its associated namespace shall be "urn:xmpp:tmp:io-data"; upon advancement of this specification, the XMPP Registrar [8] shall issue a permanent namespace in accordance with the process defined in Section 4 of XMPP Registrar Function (XEP-0053) [9].

10 XML Schema

```
<?xml version='1.0' encoding='UTF-8'?>

<xs:schema
    xmlns:xs='http://www.w3.org/2001/XMLSchema'
    targetNamespace='urn:xmpp:tmp:io-data'
    xmlns='urn:xmpp:tmp:io-data'
    elementFormDefault='qualified'>

  <xs:element name='iodata'>
    <xs:complexType>
      <xs:element name='desc' minOccurs='0' maxOccurs='1'
                  type='xs:string'/>
      <xs:element name='in' minOccurs='0' maxOccurs='1'/>
      <xs:element name='out' minOccurs='0' maxOccurs='1'/>
      <xs:element name='error' minOccurs='0' maxOccurs='1'/>
      <xs:element name='status' minOccurs='0' maxOccurs='1'/>
      <xs:attribute name='type' use='required'>
        <xs:simpleType>
          <xs:restriction base='xs:NCName'>
            <xs:enumeration value='io-schemata-get'/>
            <xs:enumeration value='io-schemata-result'/>
```

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

[8] The XMPP Registrar maintains a list of reserved protocol namespaces as well as registries of parameters used in the context of XMPP extension protocols approved by the XMPP Standards Foundation. For further information, see <https://xmpp.org/registrar/>.

[9] XEP-0053: XMPP Registrar Function <https://xmpp.org/extensions/xep-0053.html>.

```
            <xs:enumeration value='input'/>
            <xs:enumeration value='output'/>
            <xs:enumeration value='getStatus'/>
            <xs:enumeration value='getOutput'/>
            <xs:enumeration value='error'/>
            <xs:enumeration value='status'/>
          </xs:restriction>
        </xs:simpleType>
      </xs:attribute>
      <xs:anyAttribute/>
    </xs:complexType>
  </xs:element>

  <xs:element name='in'>
    <xs:choice>
      <!-- for x@type='io-schemata-result' : -->
      <xs:any namespace='http://www.w3.org/2001/XMLSchema'
              minOccurrence='1' maxOccurence='1'/>
      <!-- for x@type='input' : -->
      <xs:any namespace='##other' processContents='strict'/>
    </xs:choice>
  </xs:element>

  <xs:element name='out'>
    <xs:choice>
      <!-- for x@type='io-schemata-result' : -->
      <xs:any namespace='http://www.w3.org/2001/XMLSchema'
              minOccurrence='1' maxOccurence='1'/>
      <!-- for x@type='output' : -->
      <xs:any namespace='##other' processContents='strict'/>
    </xs:choice>
  </xs:element>

  <xs:element name='error'>
    <xs:choice>
      <!-- for x@type='io-schemata-result' : -->
      <xs:any namespace='http://www.w3.org/2001/XMLSchema'
              minOccurrence='1' maxOccurence='1'/>
      <!-- for x@type='error' : -->
      <xs:any namespace='##other' processContents='strict'/>
    </xs:choice>
  </xs:element>

  <xs:element name='status'>
    <!-- for x@type='status' -->
    <xs:element name='elapsed' minOccurs='0' maxOccurs='1'
              type='xs:integer'/>
    <xs:element name='remaining' minOccurs='0' maxOccurs='1'
              type='xs:integer'/>
```

```
    <xs:element name='percentage' minOccurs='0' maxOccurs='1'
                type='xs:integer'/>
    <xs:element name='information' minOccurs='0' maxOccurs='1'
                type='xs:string'/>
  </xs:element>

</xs:schema>
```

11 Acknowledgements

The Bioclipse Project