Network Working Group
Internet-Draft
Expires: May 4, 2003

J. Miller
P. Saint-Andre
Jabber Software Foundation

November 03, 2002

XMPP Core
draft-miller-xmpp-core-02

Status of this Memo

Copyright Notice

Abstract

This document describes the core features of the eXtensible Messaging and Presence Protocol (XMPP), which is used by the servers, clients, and other applications that comprise the Jabber network.

Table of Contents

1. Introduction

1.1 Overview

The eXtensible Messaging and Presence Protocol (XMPP) is an open, XML [1] protocol for near-real-time messaging and presence. Currently, there exist multiple implementations of the protocol, mostly offered under the name of Jabber. In addition, there are countless deployments of these implementations, which provide instant messaging (IM) and presence services at and among thousands of domains to a user base that is estimated at over one million end users. The current document defines the core constituents of XMPP; XMPP IM [2] defines the extensions necessary to provide basic instant messaging and presence functionality that addresses the requirements defined in RFC 2779 [3].

1.2 Conventions Used in this Document

The capitalized key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in RFC 2119 [4].

1.3 Discussion Venue

The authors welcome discussion and comments related to the topics presented in this document, preferably on the "xmppwg@jabber.org" mailing list (archives and subscription information are available at http://www.jabber.org/cgi-bin/mailman/listinfo/xmppwg/).

1.4 Intellectual Property Notice

This document is in full compliance with all provisions of Section 10 of RFC 2026. Parts of this specification use the term "jabber" for identifying namespaces and other protocol syntax. Jabber[tm] is a registered trademark of Jabber, Inc. Jabber, Inc. grants permission to the IETF for use of the Jabber trademark in association with this specification and its successors, if any.

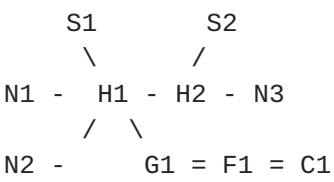
2. Generalized Architecture

2.1 Overview

Although XMPP is not wedded to any specific network architecture, to this point it has usually been implemented via a typical client-server architecture, wherein a client utilizing XMPP accesses a server over a TCP [5] socket. While it can be helpful to keep that specific architecture in mind when seeking to understand XMPP, we have herein abstracted from any specific architecture and have described the architecture in a more generalized fashion.

The following diagram provides a high-level overview of this generalized architecture (where "-" represents communications that use XMPP and "=" represents communications that use any other protocol).

Connection Map



The symbols are as follows:

- N1, N2, N3 -- Nodes on the Jabber network
- H1, H2 -- Hosts on the Jabber network
- S1, S2 -- Services that add functionality to a primary host
- G1 -- A gateway that translates between XMPP and the protocol(s) used on a foreign messaging network
- F1 -- A foreign messaging network
- C1 -- A client on a foreign messaging network

2.2 Host

A host acts as an intelligent abstraction layer for XMPP communications. Its primary responsibilities are to manage connections from or sessions for other entities (in the form of XML streams to and from authorized nodes, trusted services, and other hosts) and to route appropriately-addressed XML data "chunks" among

such entities over XML streams. Most XMPP-compliant hosts also assume responsibility for the storage of data that is used by nodes or services (e.g., the contact list for each IM user); in this case, the XML data is processed directly by the host itself on behalf of the node or service and is not routed to another entity.

2.3 Node

Most nodes connect directly to a host over a TCP socket and use XMPP to take full advantage of the functionality provided by a host and its associated services. (Clients on foreign messaging networks may also be part of the architecture, made accessable via a gateway to that network.) Multiple resources (e.g., devices or locations) MAY connect simultaneously to a host on behalf of each authorized node, with each resource connecting over a discrete TCP socket and differentiated by the resource identifier of a JID (Section 3) (e.g., node@host/home vs. node@host/work). The port assigned by the IANA [6] for connections between a Jabber node and a Jabber host is 5222. For further details about node-to-host communications for the purpose of instant messaging and presence, refer to XMPP IM [2].

2.4 Service

In addition to the basic functionality provided by a host, additional functionality is made possible by connecting trusted services to a host. Examples include multi-user chat (a.k.a. conferencing), real-time alert systems, custom authentication modules, database connectivity, and translation to foreign messaging protocols. There is no set port on which services communicate with hosts; this is left up to the administrator of the service or host. Communications between services and hosts are not defined in this document.

2.4.1 Gateway

A gateway is a special-purpose service whose primary function is to translate XMPP into the protocol(s) of another messaging system, as well as to translate the return data back into XMPP. Examples are gateways to Internet Relay Chat (IRC), Short Message Service (SMS), SMTP, and foreign instant messaging networks such as Yahoo!, MSN, ICQ, and AIM. Communications between gateways and hosts, and between gateways and the foreign messaging system, are not defined in this document.

2.5 Network

Because each host is identified by a network address (typically a DNS hostname) and because host-to-host communications are a simple extension of the node-to-host protocol, in practice the system

consists of a network of hosts that inter-communicate. Thus node-a@host1 is able to exchange messages, presence, and other information with node-b@host2. This pattern is familiar from messaging protocols (such as SMTP) that make use of network addressing standards. The usual method for providing a connection between two hosts is to open a TCP socket on the IANA-assigned port 5269 and to negotiate a connection using the Dialback Protocol (Section 5.2) as defined in this document.

3. Addressing Scheme

3.1 Overview

Any entity that can be considered a network endpoint (i.e., an ID on the network) and that can communicate using XMPP is considered a Jabber Entity. All such entities are uniquely addressable in a form that is consistent with RFC 2396 [7]. In particular, a valid Jabber Identifier (JID) contains a set of ordered elements formed of a domain identifier, node identifier, and resource identifier in the following format: [node@]domain[/resource].

All JIDs are based on the foregoing structure. The most common use of this structure is to identify an IM user, the host to which the user connects, and the user's active session or connection in the form of user@host/resource. However, other nodes are possible; for example, a specific conference room is offered by a multi-user chat service is addressed as room@service, where "room" is the name of the room and "service" is the hostname of the chat service.

3.2 Domain Identifier

The domain identifier is the primary identifier and is the only required element of a JID (a simple domain identifier is a valid JID). It usually represents the network gateway or "primary" host to which other entities connect for XML routing and data management capabilities. However, the entity referenced by a domain identifier is not always a host, and may be a service that is addressed as a subdomain of a host and that provides functionality above and beyond the capabilities of a host (a multi-user chat service, a user directory, a gateway to a foreign messaging system, etc.).

The domain identifier for every host or service that will communicate over a network SHOULD resolve to a Fully Qualified Domain Name, and a domain identifier SHOULD conform to RRC 952 [8] and REF 1123 [9]. Specifically, a domain identifier is case-insensitive 7-bit ASCII and is limited to 255 bytes.

3.3 Node Identifier

The node identifier is an optional secondary identifier. It usually represents the entity requesting and using network access provided by the host (e.g., a client), although it can also represent other kinds of entities (e.g., a multi-user chat room associated with a conference service). The entity represented by a node identifier is addressed within the context of a specific domain (e.g., user@host). Node identifiers are restricted to 256 bytes. A node identifier may contain any Unicode character higher than #x20 with the exception of

the following:

- #x22 (")
- #x26 (&)
- #x27 (')
- #x3A (:)
- #x3C (<)
- #x3E (>)
- #x40 (@)
- #x7F (del)
- #xFFFE (BOM)
- #xFFFF (BOM)

Case is preserved, but comparisons are made in case-normalized canonical form.

3.4 Resource Identifier

The resource identifer is an optional third identifier. It represents a specific session, connection (e.g., a device or location), or object (e.g., a participant in a multi-user chat room) belonging to a node. A node may maintain multiple resources simultaneously. A resource identifier is restricted to 256 bytes in length. A resource identifier MAY include any Unicode character greater than #x20, except #xFFFE and #xFFFF; if the Unicode character is a valid XML character as defined in Section 2.2 of [1], it MUST be suitably escaped for inclusion within an XML stream. Resource identifiers are case sensitive.

3.5 URIs

Full conformance with [7] would be valuable. This would most likely be effected through use of an 'xmpp:' URI scheme of the following form:

<xmpp>:[<node-identifier>@]<domain-identifier>[?<query>]

At a minimum, the 'message' and 'presence' query types would be defined, with the likely addition of query types for 'subscribe' (to

manage a subscription to teh presence of another entity) and 'roster' (to manage the representation of another entity in one's contact list). However, the use of such URIs has not yet been standardized.

4. XML Streams

4.1 Overview

Two fundamental concepts make possible the rapid, asynchronous exchange of relatively small payloads of structured information between presence-aware entities: XML streams and, as a result, discrete units of structured information that are referred to as "XML chunks". (Note: in this overview we use the example of communications between a node and host; however XML streams are more generalized and may be used for communications among hosts and services as well.)

In order to connect to a host, a node must initiate an XML stream by sending a <stream> tag to the host, optionally preceded by a text declaration specifying the XML version supported and the character encoding. A compliant entity must accept any namespace prefix on the <stream/> element; however, for historical reasons some entities may accept only a 'stream' prefix, resulting in use of a <stream:stream/> element. The host should then reply with a second XML stream back to the node, again optionally preceded by a text declaration.

Within the context of an XML stream, a sender may send a discrete semantic unit of structured information to any recipient. This unit of structured information is a well-balanced XML chunk, such as a message, presence, or IQ chunk (a chunk of an XML document is said to be well-balanced if it matches production [43] content of [1]). These chunks exist at the direct child level (depth=1) of the root <stream/> element. The start of any XML chunk is unambiguously denoted by the element start tag at depth=1 (e.g., <presence>), and the end of any XML chunk is unambiguously denoted by the corresponding close tag at depth=1 (e.g., </presence>). Each XML chunk may contain child elements or CDATA sections as necessary in order to convey the desired information from the sender to the recipient. The session is closed at the node's request by sending a closing </stream> tag to the host.

Thus a node's session with a host can be seen as two open-ended XML documents that are built up through the accumulation of the XML chunks that are sent over the course of the session (one from the node to the host and one from the host to the node), and the root <stream/> element may be considered the document entity for those streams. In essence, then, an XML stream acts as an envelope for all the XML chunks sent during a session. We can represent this graphically as follows:

```
|--------------------|
| <stream>           |
|--------------------|
| <message to=''>    |
|   <body/>          |
| </message>         |
|--------------------|
| <presence to=''>   |
|   <show/>          |
| </presence>        |
|--------------------|
| <iq to=''>         |
|   <query/>         |
| </iq>              |
|--------------------|
| </stream>          |
|--------------------|
```

4.2 Restrictions

XML streams are used to transport a subset of XML. Specifically, XML streams SHOULD NOT contain processing instructions, non-predefined entities (as defined in Section 4.6 of [1]), comments, or DTDs. Any such XML data SHOULD be ignored.

4.3 Stream Attributes

The attributes of the stream element are as follows (we now generalize the endpoints by using the terms "initiating entity" and "receiving entity"):

- to -- The 'to' attribute should be used only in the XML stream from the initiating entity to the receiving entity, and must be set to the JID of the receiving entity. There should be no 'to' attribute set in the XML stream by which the receiving entity replies to the initiating entity; however, if a 'to' attribute is included, it SHOULD be ignored by the receiving entity.

- from -- The 'from' attribute should be used only in the XML stream from the receiving entity to the initiating entity, and must be set to the JID of the receiving entity granting access to the initiating entity. There should be no 'from' attribute on the XML stream sent from the initiating entity to the receiving entity; however, if a 'from' attribute is included, it SHOULD be ignored by the receiving entity.

- id -- The 'id' attribute should be used only in the XML stream

from the receiving entity to the initiating entity. This attribute is a unique identifier created by the receiving entity to function as a session key for the initiating entity's session with the receiving entity. There should be no 'id' attribute on the XML stream sent from the initiating entity to the receiving entity; however, if an 'id' attribute is included, it SHOULD be ignored by the receiving entity.

We can summarize these values as follows:

	initiating to receiving	receiving to initiating
to	JID of receiver	ignored
from	ignored	JID of receiver
id	ignored	session key

4.4 Namespace Declarations

The stream element may also contain namespace declarations as defined in [11].

A stream namespace declaration is REQUIRED in both XML streams. A compliant entity must accept any namespace prefix on the <stream/> element; however, for historical reasons some entities may accept only a 'stream' prefix, resulting in use of a <stream:stream/> element as the stream root. The value of the stream namespace MUST be "http://etherx.jabber.org/streams".

A default namespace declaration ('xmlns') is REQUIRED and is used in both XML streams in order to scope the allowable first-level children of the root stream element for both streams. This namespace declaration must be the same for the initiating stream and the responding stream so that both streams are scoped consistently.

XML streams function as containers for any XML chunks sent asynchronously between network endpoints. It should be possible to scope an XML stream with any default namespace declaration, i.e., it should be possible to send any properly-namespaced XML chunk over an XML stream. However, for historical reasons existing implementations will support only the following default namespaces:

- jabber:client -- this default namespace is declared when the stream is used for communications between a node and a host

- jabber:server -- this default namespace is declared when the stream is used for communications between two hosts

- jabber:component:accept or jabber:component:connect -- one of these default namespaces is declared when the stream is used for communications between a host and a trusted service

This document addresses the jabber:client and jabber:server namespaces only (indeed these two namespaces have identical schemas). The jabber:component:* namespaces are outside the scope of this document.

4.5 Stream Errors

The root stream element MAY contain an error child element (e.g., <stream:error/> if the stream namespace prefix is 'stream'). The error child is used to signify that a stream-level error has occurred. Examples include the sending of invalid XML, the shutdown of a host, an internal server error such as the shutdown of a session manager, and an attempt by a node to authenticate as the same resource that is currently connected. If an error occurs at the level of the stream, the entity (initiating entity or receiving entity) that detects the error should send a stream error to the other entity specifying why the streams are being closed and then send a closing </stream> tag. XML of the following form is sent within the context of an existing stream:

```
<stream:stream ...>
...
<stream:error>
  Error message (e.g., "Invalid XML")
</stream:error>
</stream:stream>
```

4.6 Example

The following is a simple stream-based session of a node on a host (where the NODE lines are sent from the node to the host, and the HOST lines are sent from the host to the node):

A simple session:

```
NODE: <?xml version='1.0'?>
      <stream:stream
          to='host'
          xmlns='jabber:client'
          xmlns:stream='http://etherx.jabber.org/streams'>
HOST: <?xml version='1.0'?>
      <stream:stream
          from='host'
          id='id_123456789'
          xmlns='jabber:client'
          xmlns:stream='http://etherx.jabber.org/streams'>
NODE:   <message from='node@host' to='receiving-ID'>
NODE:     <body>Watson come here, I need you!</body>
NODE:   </message>
HOST:   <message from='receiving-ID' to='node@host'>
HOST:     <body>I'm on my way!</body>
HOST:   </message>
NODE: </stream:stream>
HOST: </stream:stream>
```

These are in actuality a sending stream and a receiving stream, which can be viewed a-chronologically as two XML documents:

```
NODE: <?xml version='1.0'?>
      <stream:stream
          to='host'
          xmlns='jabber:client'
          xmlns:stream='http://etherx.jabber.org/streams'>
NODE:   <message from='node@host' to='receiving-ID'>
NODE:     <body>Watson come here, I need you!</body>
NODE:   </message>
NODE: </stream:stream>

HOST: <?xml version='1.0'?>
      <stream:stream
          from='host'
          id='id_123456789'
          xmlns='jabber:client'
          xmlns:stream='http://etherx.jabber.org/streams'>
HOST:   <message from='receiving-ID' to='node@host'>
HOST:     <body>I'm on my way!</body>
HOST:   </message>
HOST: </stream:stream>
```

A session gone bad:

```
NODE: <?xml version='1.0'?>
      <stream:stream
          to='host'
          xmlns='jabber:client'
          xmlns:stream='http://etherx.jabber.org/streams'>
HOST: <?xml version='1.0'?>
      <stream:stream
          from='host'
          id='id_123456789'
          xmlns='jabber:client'
          xmlns:stream='http://etherx.jabber.org/streams'>
NODE: <message><body>Bad XML, no closing body tag!</message>
HOST: <stream:error>Invalid XML</stream:error>
HOST: </stream:stream>
```

5. Stream Authentication

XMPP includes two methods for enforcing authentication at the level of XML streams. When one entity is already known to another (i.e., there is an existing trust relationship between the entities such as that established when a node registers with a host or an administrator configures a host to trust a service), the preferred method for authenticating streams between the two entities uses an XMPP adaptation of the Simple Authentication and Security Layer (SASL) [10]. When there is no existing trust relationship between the two entities, such trust MAY be established based on existing trust in DNS; the authentication method used when two such entities are hosts is the server dialback protocol that is native to XMPP. Both of these methods are described in this section.

5.1 SASL Authentication

5.1.1 Overview

The Simple Authentication and Security Layer (SASL) provides a generalized method for adding authentication support to connection-based protocols. XMPP uses a generic XML namespace profile for SASL that conforms to section 4 ("Profiling Requirements") of [10] (the namespace identifier for this protocol is http://www.iana.org/assignments/sasl-mechanisms). If an entity (node, host, or service) is capable of authenticating by means of SASL, it MUST include the agreed-upon SASL namespace within the opening root stream tag it uses to initiate communications.

The following example shows the use of SASL in node authentication with a host, for which the steps involved are as follows:

1. The node requests SASL authentication by including the appropriate namespace declaration (xmlns:sasl='http://www.iana.org/assignments/sasl-mechanisms') in the opening XML stream header sent to the host.

2. The host includes the xmlns:sasl namespace declaration in the XML stream header sent in reply to the node.

3. The host responds with a list of available SASL authentication mechanisms, each of which is a <mechanism/> element included as a child within a <mechanisms/> container element that is sent as a first-level child of the root <stream/> element.

4. The node selects a mechanism by sending a <sasl:auth/> element to the host; this element MAY optionally contain character data.

5. If necessary, the host challenges the node by sending a <sasl:challenge/> element to the node; this element MAY optionally contain character data.

6. The node responds to challenge by sending a <sasl:response/> element to the host; this element MAY optionally contain character data.

7. If necessary, the host sends more challenges and the node sends more responses.

This series of challenge/response pairs continues until one of three things happens:

- The node aborts the handshake by sending a <sasl:abort/> element to the host.

- The host reports failure by sending a <sasl:failure/> element to the node.

- The host reports success by sending a <sasl:success/> element to the node; this element MAY optionally contain character data.

Any character data contained within these elements MUST be encoded using base64.

5.1.2 Example

The following example shows the data flow for a node authenticating with a host using SASL.

Step 1: Node initiates stream to host:

```
<stream:stream
    xmlns='jabber:client'
    xmlns:stream='http://etherx.jabber.org/streams'
    xmlns:sasl='http://www.iana.org/assignments/sasl-mechanisms'
    to='capulet.com'
    version='1.0'>
```

Step 2: Host responds with a stream tag sent to the node:

```
<stream:stream
    xmlns='jabber:client'
    xmlns:stream='http://etherx.jabber.org/streams'
    xmlns:sasl='http://www.iana.org/assignments/sasl-mechanisms'
    id='12345678'
    version='1.0'>
```

Step 3: Host informs node of available authentication mechanisms:

```
<sasl:features>
  <mechanisms xmlns='http://www.iana.org/assignments/sasl-mechanisms'>
    <mechanism>DIGEST-MD5</mechanism>
    <mechanism>PLAIN</mechanism>
  </mechanisms>
  <starttls xmlns='http://www.ietf.org/rfc/rfc2246.txt'/>
</sasl:features>
```

Step 4: Node selects an authentication mechanism:

```
<sasl:auth mechanism='DIGEST-MD5'/>
```

Step 5: Host sends a challenge to the node:

```
<sasl:challenge>
    cmVhbG09ImNhdGFjbHlzbS5jeCIsbm9uY2U9Ik9BNk1HOXRFUUdtMmhoIi
    xxb3A9ImF1dGgiLGNoYXJzZXQ9dXRmLTgsYWxnb3JpdGhtPW1kNS1zZXNz
</sasl:challenge>
```

Step 6: Node responds to the challenge:

```
<sasl:response>
    dXNlcm5hbWU9InJvYiIscmVhbG09ImNhdGFjbHlzbS5jeCIsbm9uY2U9Ik
    9BNk1HOXRFUUdtMmhoIixjbm9uY2U9Ik9BNk1IWGg2VnFUclJrIixuYz0w
    MDAwMDAwMSxxb3A9YXV0aCxkaWdlc3QtdXJpPSJqYWJiZXIvY2F0YWNseX
    NtLmN4IixyZXNwb25zZT1kMzg4ZGFkOTBkNGJiZDc2MGExNTIzMjFmMjE0
    M2FmNyxjaGFyc2V0PXV0Zi04
</sasl:response>
```

Step 7: Host sends another challenge to the node:

```
<sasl:challenge>
    cnNwYXV0aD1lYTQwZjYwMzM1YzQyN2I1NTI3Yjg0ZGJhYmNkZmZmZA==
</sasl:challenge>
```

Step 8: Node responds to the challenge:

<sasl:response/>

Step 9: Host informs node of successful authentication:

<sasl:success/>

Step 9 (alt): Host informs node of failed authentication:

<sasl:failure/>

5.2 Dialback Authentication

XMPP includes a protocol-level method for verifying that a connection between two hosts may be trusted. The method is called dialback and is used only within XML streams that are declared under the "jabber:server" namespace.

The purpose of the dialback protocol is to make server spoofing more difficult, and thus to make it more difficult to forge XML chunks. Dialback is not intended as a mechanism for securing or encrypting the streams between servers, only for helping to prevent the spoofing of a hostname and the sending of false data from it. Dialback is made possible by the existence of DNS, since one host can verify that another host which is connecting to it is authorized to represent a given host on the Jabber network. All DNS host resolutions must first resolve the host using an SRV [12] record of _jabber._tcp.host. If the SRV lookup fails, the fallback is a normal A lookup to determine the IP address, using the jabber-server port of 5269 assigned by the Internet Assigned Numbers Authority [6].

Note that the method used to generate and verify the keys used in the dialback protocol must take into account the hostnames being used, along with a secret known only by the receiving host and the random id per stream. Generating unique but verifiable keys is important to prevent common man-in-the-middle attacks and host spoofing.

In the description that follows we use the following terminology:

- Originating Host -- the host that is attempting to establish a connection between the two hosts
- Receiving Host -- the host that is trying to authenticate that the Originating Host represents the Jabber host which it claims to be
- Authoritative Host -- the host which is given when a DNS lookup is

performed on the name that the Originating Host initially gave; for simple environments this will be the Originating Host, but it could be a separate machine in the Originating Host's network

The following is a brief summary of the order of events in dialback:

1. Originating Host establishes a connection to Receiving Host.

2. Originating Host sends a 'key' value over the connection to Receiving Host.

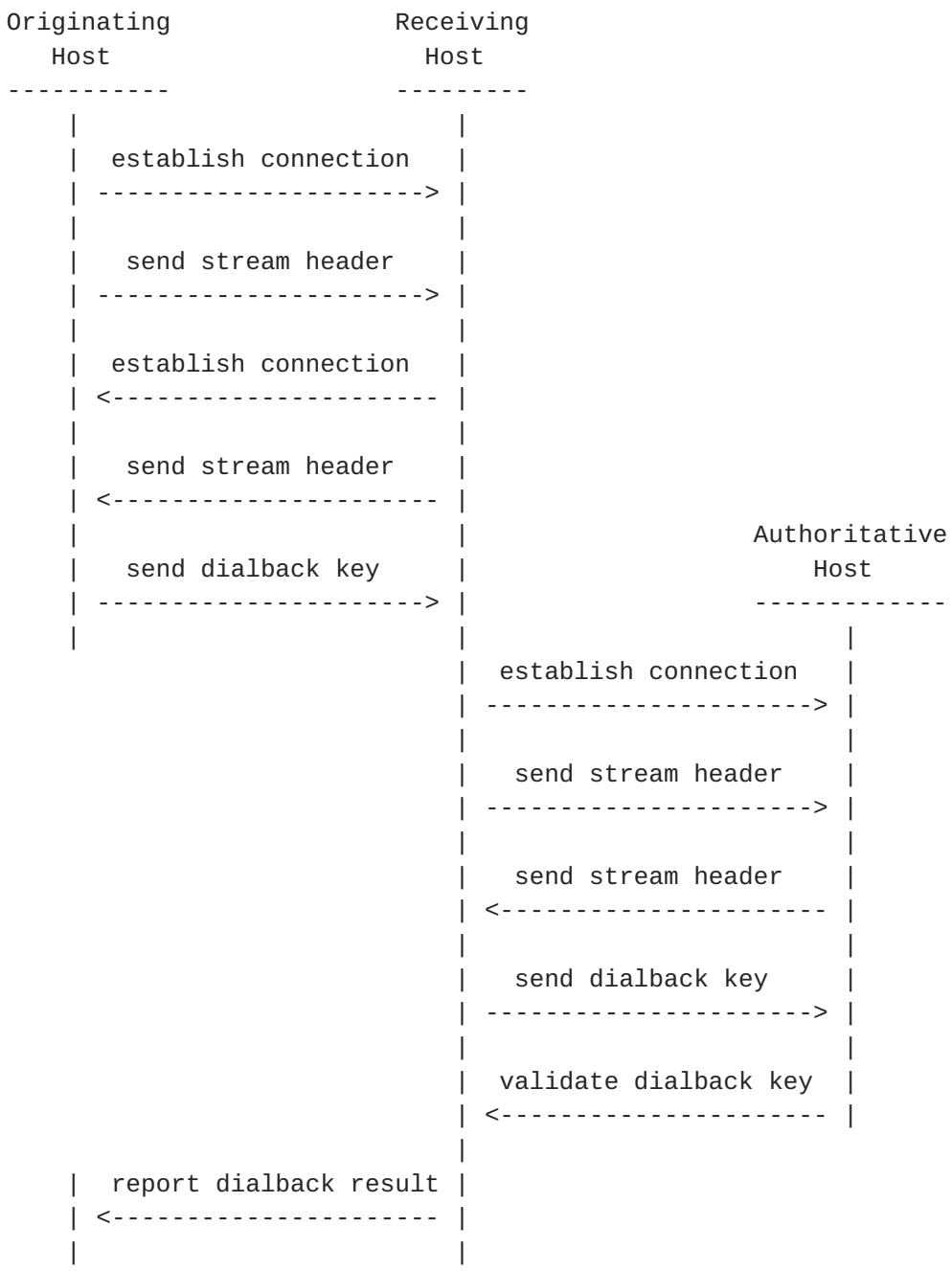
3. Receiving Host establishes a connection to Authoritative Host.

4. Receiving Host sends the same 'key' value to Authoritative Host.

5. Authoritative Host replies that key is valid or invalid.

6. Receiving Host tells Originating Host whether it is authenticated or not.

We can represent this flow of events graphically as follows:



5.2.1 Dialback Protocol

The traffic sent between the hosts is as follows:

1. Originating Host establishes connection to Receiving Host

2. Originating Host sends a stream header to Receiving Host (the 'to' and 'from' attributes are not required):

```
<stream:stream
    xmlns:stream='http://etherx.jabber.org/streams'
    xmlns='jabber:server'
    xmlns:db='jabber:server:dialback'>
```

Note: the value of the xmlns:db namespace declaration indicates to Receiving Host that the Originating Host supports dialback.

3. Receiving Host sends a stream header back to Originating Host (the 'to' and 'from' attributes are not required):

```
<stream:stream
    xmlns:stream='http://etherx.jabber.org/streams'
    xmlns='jabber:server'
    xmlns:db='jabber:server:dialback'
    id='457F9224A0...'>
```

4. Originating Host sends a dialback key to Receiving Host:

```
<db:result
    to='Receiving Host'
    from='Originating Host'>
  98AF014EDC0...
</db:result>
```

Note: this key is not examined by Receiving Host, since the Receiving Host does not keep information about Originating Host between sessions.

5. Receiving Host now establishes a connection back to Originating Host, getting the Authoritative Host.

6. Receiving Host sends Authoritative Host a stream header (the 'to' and 'from' attributes are not required):

```
<stream:stream
    xmlns:stream='http://etherx.jabber.org/streams'
    xmlns='jabber:server'
    xmlns:db='jabber:server:dialback'>
```

7. Authoritative Host sends Receiving Host a stream header:

```
<stream:stream
    xmlns:stream='http://etherx.jabber.org/streams'
    xmlns='jabber:server'
    xmlns:db='jabber:server:dialback'
    id='1251A342B...'>
```

8. Receiving Host sends Authoritative Host a chunk indicating it wants Authoritative Host to verify a key:

```
<db:verify
    from='Receiving Host'
    to='Originating Host'
    id='457F9224A0...'>
  98AF014EDC0...
</db:verify>
```

Note: passed here are the hostnames, the original identifier from Receiving Host's stream header to Originating Host in step 2, and the key Originating Host gave Receiving Host in step 3. Based on this information and shared secret information within the 'Originating Host' network, the key is verified. Any verifiable method can be used to generate the key.

9. Authoritative Host sends a chunk back to Receiving Host indicating whether the key was valid or invalid:

```
<db:result
    from='Originating Host'
    to='Receiving Host'
    type='valid'
    id='457F9224A0...'/>
```

or

```
<db:result
    from='Originating Host'
    to='Receiving Host'
    type='invalid'
    id='457F9224A0...'/>
```

10. Receiving Host informs Originating Host of the result:

```
<db:result
    from='Receiving Host'
    to='Originating Host'
    type='valid'/>
```

Note: At this point the connection has either been validated via a type='valid', or reported as invalid. Once the connection is validated, data can be sent by the Originating Host and read by the Receiving Host; before that, all data chunks sent to Receiving Host SHOULD be dropped. As a final guard against domain spoofing, the Receiving Host MUST verify that all XML chunks received from the Originating Host include a 'from'

attribute and that from address of each chunk includes the validated domain. In addition, all XML chunks of type message, presence, and IQ MUST include a 'to' attribute.

6. Core Data Elements

6.1 Overview

The core data elements for XMPP communications are <message/>, <presence/>, and <iq/>. These data elements are sent as direct (depth=1) children of the root <stream/> element and are scoped by one of the default namespaces identified in Section 4.4.

6.2 Common Attributes

Four attributes are common to message, presence, and IQ chunks. These are defined below.

6.2.1 to

The 'to' attribute specifies the JID of the intended recipient for the chunk. A chunk SHOULD possess a 'to' attribute. A chunk sent from a node to a host for handling by that host (e.g., presence sent to the host for broadcasting to other entities) MAY legitimately lack a 'to' attribute.

6.2.2 from

The 'from' attribute specifies the JID of the sender.

A node MUST NOT include a 'from' attribute on the chunks it sends to a host; if a host receives a chunk from a node and the chunk possesses a 'from' attribute, it must ignore the value of the 'from' attribute. A host MUST stamp chunks received from a node with the user@host/resource (full JID) of the connected resource that generated the chunk.

A host MUST include a 'from' attribute on chunks it routes to other hosts. The domain identifier of the JID contained in the 'from' attribute MUST match the hostname of the host as communicated in the dialback negotiation (or a subdomain thereof).

6.2.3 id

The optional 'id' attribute may be used to track chunks sent and received. The 'id' attribute is generated by the sender. An 'id' attribute included in an IQ request of type "get" or "set" SHOULD be returned to the sender in any IQ response of type "result" or "error" generated by the recipient of the request. A recipient of a message or presence chunk MAY return that 'id' in any replies, but is not required to do so.

6.2.4 type

The 'type' attribute specifies detailed information about the purpose or context of the message, presence, or IQ chunk. The particular allowable values for the 'type' attribute vary depending on whether the chunk is a message, presence, or IQ, and thus are specified in the following sections.

6.3 Message Chunks

Message chunks in the 'jabber:client' or 'jabber:server' namespace are used to "push" information to another entity. Common uses in the context of instant messaging include single messages, messages sent in the context of a chat conversation, messages sent in the context of a multi-user chat room, headlines, and errors. These messages types are identified more fully below.

6.3.1 Types of Message

The 'type' attribute of a message chunk is optional and specifies the conversational context of the message. The sending of a message chunk without a 'type' attribute signals that the message chunk is a single message. However, the 'type' attribute may also have one of the following values:

- chat -- The message is sent in the context of a one-to-one chat conversation.
- groupchat -- The message is sent in the context of a multi-user chat environment.
- headline -- The message is generated by an automated service that delivers content (news, sports, market information, etc.).
- error - A message returned to a sender specifying an error associated with a previous message sent by the sender (for a full list of error messages, see error codes (Appendix A))

For detailed information about these message types, refer to XMPP IM [2].

6.3.2 Children

If a message chunk in the 'jabber:client' or 'jabber:server' namespace has no 'type' attribute or has a 'type' attribute with a value of "chat", "groupchat", or "headline", it MAY contain zero or one of each of the following child elements (which MUST NOT contain mixed content):

- body -- The textual contents of the message; normally included but not required. The <body/> element MUST NOT have any attributes.

- subject -- The subject of the message. The <subject/> element MUST NOT have any attributes.

- thread -- A random string that is generated by the sender and that MAY be copied back in replies; it is used for tracking a conversation thread. The <thread/> element MUST NOT have any attributes.

If the message chunk is of type "error", it MUST include an <error/> child, which in turn MUST possess a 'code' attribute corresponding to one of the standard error codes (Appendix A) and MAY also contain PCDATA corresponding to a natural-language description of the error. An <error/> child MUST NOT be included if the chunk type is anything other than "error".

As described under extended namespaces (Section 6.6), a message chunk MAY also contain any properly-namespaced child element (other than the core data elements, stream elements, or defined children thereof).

6.4 Presence Chunks

Presence chunks are used in the 'jabber:client' or 'jabber:server' namespace to express an entity's current availability status (offline or online, along with various sub-states of the latter) and to communicate that status to other entities. They are also used to negotiate and manage subscriptions to the presence of other entities.

6.4.1 Types of Presence

The 'type' attribute of a presence chunk is optional. A presence chunk that does not have a 'type' attribute is used to signal that the sender is online and available for communication. If included, the 'type' attribute specifies the availability state of the sender, a request to manage a subscription to another entity's presence, a request for another entity's current presence, or an error related to a previously-sent presence chunk. The 'type' attribute may have one of the following values:

- unavailable -- Signals that the entity is no longer available for communication.

- subscribe -- The sender wishes to subscribe to the recipient's presence.

- subscribed -- The sender has allowed the recipient to receive their presence.

- unsubscribe -- A notification that an entity is unsubscribing from another entity's presence.

- unsubscribed -- The subscription request has been denied or a previously-granted subscription has been cancelled.

- probe -- A request for an entity's current presence.

- error -- An error has occurred regarding processing or delivery of a previously-sent presence chunk.

Information about the subscription model used within XMPP may be found in [2].

6.4.2 Children

If a presence chunk possesses no 'type' attribute, it MAY contain zero or one of each of the following child elements (for historical reasons the <status/> child MAY be sent in a presence chunk of type "subscribe"):

- show -- Describes the availability status of an entity or specific resource. The value SHOULD be one of the following (values other than these four MAY be ignored; additional availability types should be defined through a properly-namespaced child element of the presence chunk):

  - away -- The entity or resource is temporarily away.

  - chat -- The entity or resource is actively interested in chatting.

  - xa -- The entity or resource is away for an extended period (xa = "eXtended Away").

  - dnd -- The entity or resource is busy (dnd = "Do Not Disturb").

- status -- An optional natural-language description of availability status. Normally used in conjunction with the show element to provide a detailed description of an availability state (e.g., "In a meeting").

- priority -- A non-negative integer representing the priority level of the connected resource, with zero as the lowest priority.

If the presence chunk is of type "error", it MUST include an <error/> child, which in turn MUST possess a 'code' attribute corresponding to one of the standard error codes (Appendix A) and MAY also contain PCDATA corresponding to a natural-language description of the error. An <error/> child MUST NOT be included if the chunk type is anything other than "error".

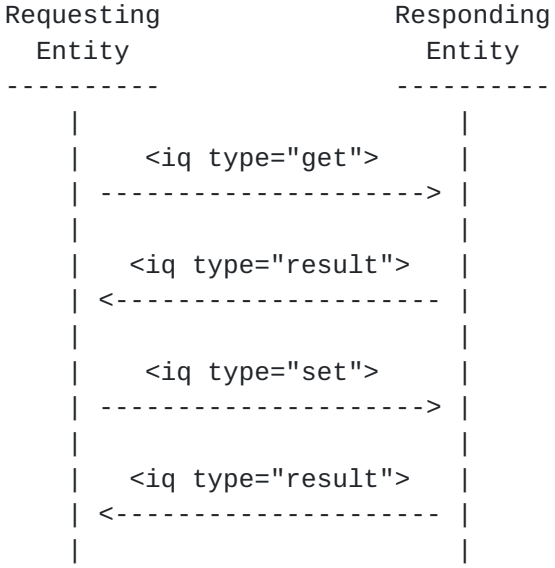
As described under extended namespaces (Section 6.6), a presence chunk MAY also contain any properly-namespaced child element (other than the core data elements, stream elements, or defined children thereof).

6.5 IQ Chunks

6.5.1 Overview

Info/Query, or IQ, is a simple request-response mechanism. Just as HTTP is a request-response medium, IQ chunks in the 'jabber:client' or 'jabber:server' namespace enable an entity to make a request of, and receive a response from, another entity. The data content of the request and response is defined by the namespace declaration of a direct child element of the iq element.

Most IQ interactions follow a common pattern of structured data exchange such as get/result or set/result:



6.5.2 Types of IQ

The 'type' attribute of an IQ chunk is REQUIRED. The 'type' attribute specifies a distinct step within a request-response

interaction. The value SHOULD be one of the following (all other values MAY be ignored):

- get -- The chunk is a request for information.
- set -- The chunk provides required data, sets new values, or replaces existing values.
- result -- The chunk is a response to a successful get or set request.
- error -- An error has occurred regarding processing or delivery of a previously-sent get or set.

6.5.3 Children

An IQ chunk contains no children in the 'jabber:client' or 'jabber:server' namespace since it is a vessel for XML in another namespace. As described under extended namespaces (Section 6.6), an IQ chunk MAY contain any properly-namespaced child element (other than the core data elements, stream elements, or defined children thereof).

If the IQ chunk is of type "error", it MUST include an <error/> child, which in turn MUST possess a 'code' attribute corresponding to one of the standard error codes (Appendix A) and MAY also contain PCDATA corresponding to a natural-language description of the error. An <error/> child MUST NOT be included if the chunk type is anything other than "error".

6.6 Extended Namespaces

While the core data elements defined in this document provide a basic level of functionality for messaging and presence, XMPP uses XML namespaces to extend the core data elements for the purpose of providing additional functionality. Thus a message, presence, or IQ chunk may house one or more optional child elements containing content that extends the meaning of the message (e.g., an encrypted form of the message body as described in Appendix C). This child element MAY be any element (other than the core data elements, stream elements, or defined children thereof). The child element MUST possess an 'xmlns' namespace declaration (other than the stream namespace and the default namespace) that defines all data contained within the child element.

Support for any given extended namespace is OPTIONAL on the part of any implementation. If an entity does not understand such a

namespace, it must ignore the associated XML data. If an entity receives an IQ chunk in a namespace it does not understand, the entity SHOULD return an IQ chunk of type "error" with an error element of code 400 (bad request). If an entity receives a message or presence chunk that contains XML data in an extended namespace it does not understand, the portion of the chunk that is in the unknown namespace SHOULD be ignored. If an entity receives a message chunk without a <body/> element but containing only a child element bound by a namespace it does not understand, it MUST ignore that chunk.

7. XML Usage within XMPP

7.1 Overview

In essence, XMPP core consists of three interrelated parts:

1. XML streams (Section 4), which provide a stateful means for transporting data in an asynchronous manner from one entity to another

2. stream authentication using SASL authentication (Section 5.1) or the dialback protocol (Section 5.2)

3. core data elements (Section 6) (message, presence, and iq), which provide a framework for communications between entities

XML [1] is used to define each of these protocols, as described in detail in the following sections.

In addition, XMPP contains protocol extensions (such as extended namespaces) that address the specific functionality required to create a basic instant messaging and presence application; these non-core protocol extensions are defined in XMPP IM [2].

7.2 Namespaces

XML Namespaces [11] are used within all XMPP-compliant XML to create strict boundaries of data ownership. The basic function of namespaces is to separate different vocabularies of XML elements that are structurally mixed together. Ensuring that XMPP-compliant XML is namespace-aware enables any XML to be structurally mixed with any data element within XMPP. This feature is relied upon frequently within XMPP to separate the XML that is processed by different services. Mainly for historical reasons, the default namespace for XMPP data chunks MUST be one of the namespaces identified in Section 4.4.

Additionally, XMPP is more strict about namespace prefixes than the XML namespace specification requires.

7.3 Validation

A host is not responsible for validating the XML elements forwarded to a node; an implementation MAY choose to provide only validated data elements but is NOT REQUIRED to do so. Nodes and services SHOULD NOT rely on the ability to send data which does not conform to the schemas, and SHOULD ignore any non-conformant elements or attributes on the incoming XML stream.

7.4 Character Encodings

Software implementing XML streams MUST support the UTF-8 and UTF-16 encodings for received data. Software MUST NOT attempt to use any other encoding for transmitted data. The encodings of the transmit and receive streams are independent. Software may select either UTF-8 or UTF-16 for the transmitted stream, and should deduce the encoding of the received stream as described in [1].

7.5 Inclusion of Text Declaration

An application MAY send a text declaration. Applications MUST follow the rules in [1] concerning the circumstances in which a text declaration is included.

8. IANA Considerations

The IANA registers "jabber-client" and "jabber-server" as GSS-API [14] service names, as specified in Section 6.1.1.

9. Internationalization Considerations

- A node SHOULD include an xml:lang declaration on the stream:stream it initiates to a host, denoting the node's default (preferred) language.

- If the host detects an xml:lang declaration on the stream:stream from a node, it SHOULD remember that value.

- If a host does not receive an xml:lang from a node, it SHOULD have a configurable default locale that it remembers instead.

- For all chunks, if the node does not send an xml:lang attribute on the root tag of the packet, the server SHOULD apply its remembered value.

- If a node does send an xml:lang attribute on a chunk, the server MUST NOT modify or delete it.

- A host SHOULD include an xml:lang declaration on the stream:stream with which it replies to a node, denoting the host's default (preferred) language.

10. Security Considerations

10.1 Node-to-Host Communications

The SASL protocol for authenticating XML streams negotiated between a node and a host (defined under Section 5.1 above) provides a reliable mechanism for validating that a node connecting to a host is who it claims to be.

The IP address and method of access of nodes MUST NOT be made available by a host, nor are any connections other than the original host connection required. This helps protect the node's host from direct attack or identification by third parties.

End-to-end encryption of message bodies and presence status information MAY be effected through use of OpenPGP [13]; for details, see Appendix C.

10.2 Host-to-Host Communications

It is OPTIONAL for any given host to communicate with other hosts, and host-to-host communications MAY be disabled by the administrator of any given deployment.

If two hosts would like to enable communications between themselves, they MUST form a relationship of trust at some level, either based on trust in DNS or based on a pre-existing trust relationship (e.g., through exchange of certificates). If two hosts have a pre-existing trust relationship, they MAY use SASL Authentication (Section 5.1) for the purpose of authenticating each other. If they do not have a pre-existing relationship, they MUST use the Dialback Protocol (Section 5.2), which provides a reliable mechanism for preventing the spoofing of hosts.

10.3 Use of SASL

Although service provisioning is a policy matter, at a minimum, all implementations MUST provide the SASL DIGEST-MD5 mechanism for authentication.

References

[1] World Wide Web Consortium, "Extensible Markup Language (XML) 1.0 (Second Edition)", W3C xml, October 2000, <http://www.w3.org/TR/2000/REC-xml-20001006>.

[2] Miller, J. and P. Saint-Andre, "XMPP Instant Messaging (draft-miller-xmpp-im-02, work in progress)", November 2002.

[3] Day, M., Aggarwal, S., Mohr, G. and J. Vincent, "A Model for Presence and Instant Messaging", RFC 2779, February 2000, <http://www.ietf.org/rfc/rfc2779.txt>.

[4] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, March 1997.

[5] University of Southern California, "Transmission Control Protocol", RFC 793, September 1981, <http://www.ietf.org/rfc/rfc0793.txt>.

[6] Internet Assigned Numbers Authority, "Internet Assigned Numbers Authority", January 1998, <http://www.iana.org/>.

[7] Berners-Lee, T., Fielding, R. and L. Masinter, "Uniform Resource Identifiers (URI): Generic Syntax", RFC 2396, August 1998, <http://www.ietf.org/rfc/rfc2396.txt>.

[8] Harrenstien, K., Stahl, M. and E. Feinler, "DoD Internet host table specification", RFC 952, October 1985.

[9] Braden, R., "Requirements for Internet Hosts - Application and Support", STD 3, RFC 1123, October 1989.

[10] Myers, J., "Simple Authentication and Security Layer (SASL)", RFC 2222, October 1997.

[11] World Wide Web Consortium, "Namespaces in XML", W3C xml-names, January 1999, <http://www.w3.org/TR/1999/REC-xml-names-19990114/>.

[12] Gulbrandsen, A. and P. Vixie, "A DNS RR for specifying the location of services (DNS SRV)", RFC 2052, October 1996.

[13] Elkins, M., Del Torto, D., Levien, R. and T. Roessler, "MIME Security with OpenPGP", RFC 3156, August 2001.

[14] Linn, J., "Generic Security Service Application Program Interface, Version 2", RFC 2078, January 1997.

Authors' Addresses

Jeremie Miller
Jabber Software Foundation
1899 Wynkoop Street, Suite 600
Denver, CO 80202
US

EMail: jeremie@jabber.org
URI: http://www.jabber.org/people/jer.php

Peter Saint-Andre
Jabber Software Foundation
1899 Wynkoop Street, Suite 600
Denver, CO 80202
US

EMail: stpeter@jabber.org
URI: http://www.jabber.org/people/stpeter.php

Appendix A. Standard Error Codes

A standard error element is used for failed processing of XML chunks. This element is a child of the failed chunk and MUST include a 'code' attribute corresponding to one of the following error codes.

- 302 (Redirect) - Whereas HTTP contains eight different codes for redirection, XMPP contains only one (which is intended to stand for any redirection error). However, code 302 is being reserved for future functionality and is not implemented at this time.

- 400 (Bad Request) - Code 400 is used to inform a sender that a request could not be understood by the recipient. This might be generated when, for example, an entity sends a message that does not have a 'to' attribute.

- 401 (Unauthorized) - Code 401 is used to inform nodes that they have provided incorrect authorization information, e.g., an incorrect password or unknown username when attempting to authenticate with a host.

- 402 (Payment Required) - Code 402 is being reserved for future use.

- 403 (Forbidden) - Code 403 is used to inform an entity that the its request was understood but that the recipient is refusing to fulfill it, e.g., if a node attempts to set information associated with another node.

- 404 (Not Found) - Code 404 is used to inform a sender that no recipient was found matching the JID to which an XML chunk was sent, e.g., if a sender has attempted to send a message to a JID that does not exist. (Note: if the host of the intended recipient cannot be reached, an error code from the 500 series must be sent).

- 405 (Not Allowed) - Code 405 is used when the action requested is not allowed for the JID identified by the 'from' address, e.g., if a node attempts to set the time or version of a host.

- 406 (Not Acceptable) - Code 406 is used when an XML chunk is for some reason not acceptable to a host or other entity. This might be generated when, for example, a node attempts to register with a host using an empty password.

- 407 (Registration Required) - Code 407 is used when a message or request is sent to a service that requires prior registration, e.g., if a node attempts to send a message through a gateway to a

foreign messaging system without having first registered with that gateway.

- 408 (Request Timeout) - Code 408 is returned when a recipient does not produce a response within the time that the sender was prepared to wait.

- 500 (Internal Server Error) - Code 500 is used when a host or service encounters an unexpected condition which prevents it from handling an XML chunk from a sender, e.g., if an authentication request is not handled by a host because the password could not be retrieved.

- 501 (Not Implemented) - Code 501 is used when the recipient does not support the functionality being requested by a sender, e.g., if a node attempts to register with a host that does not allow registration.

- 502 (Remote Server Error) - Code 502 is used when delivery of an XML chunk fails because of an inability to reach the intended remote host or service, e.g., because a remote host's hostname could not be resolved.

- 503 (Service Unavailable) - Code 503 is used when a sender requests a service that a recipient is temporarily unable to offer.

- 504 (Remote Server Timeout) - Code 504 is used when attempts to contact a remote host timeout, e.g., if an incorrect hostname is specified.

Appendix B. Formal Definitions

B.1 streams namespace

B.1.1 DTD

```
<?xml version='1.0' encoding='UTF-8'?>
<!ELEMENT stream (#PCDATA | error?)*>
<!ATTLIST stream
  to            CDATA  #REQUIRED
  from          CDATA  #IMPLIED
  id            CDATA  #IMPLIED>
<!ELEMENT error (#PCDATA)>
```

B.1.2 Schema

```
<?xml version='1.0' encoding='UTF-8'?>
<xsd:schema
    xmlns:xsd='http://www.w3.org/2001/XMLSchema'
    targetNamespace='http://etherx.jabber.org/streams'
    xmlns='http://etherx.jabber.org/streams'
    elementFormDefault='qualified'>

  <xsd:element name='stream'>
    <xsd:complexType mixed='true'>
      <xsd:element ref='error' minOccurs='0' maxOccurs='1'/>
      <xsd:choice>
        <xsd:any
             namespace='jabber:client'
             maxOccurs='1'/>
        <xsd:any
             namespace='jabber:component:accept'
             maxOccurs='1'/>
        <xsd:any
             namespace='jabber:component:connect'
             maxOccurs='1'/>
        <xsd:any
             namespace='jabber:server'
             maxOccurs='1'/>
        <xsd:any
             namespace='http://www.iana.org/assignments/sasl-mechanisms'
             maxOccurs='1'/>
      </xsd:choice>
      <xsd:attribute name='to' type='xsd:string' use='optional'/>
      <xsd:attribute name='from' type='xsd:string' use='optional'/>
      <xsd:attribute name='id' type='xsd:string' use='optional'/>
    </xsd:complexType>
  </xsd:element>

  <xsd:element name='error' type='xsd:string'/>

</xsd:schema>
```

B.2 sasl namespace

B.2.1 DTD

The DTD for the sasl: namespace is as follows:

```
<?xml version='1.0' encoding='UTF-8'?>
<!ELEMENT mechanisms (mechanism)*>
<!ELEMENT mechanism (#PCDATA)>
<!ATTLIST mechanism name CDATA #REQUIRED>
<!ELEMENT auth (#PCDATA)>
<!ATTLIST auth name CDATA #REQUIRED>
<!ELEMENT challenge (#PCDATA)>
<!ELEMENT response (#PCDATA)>
<!ELEMENT abort (#PCDATA)>
<!ELEMENT success (#PCDATA)>
<!ELEMENT failure (#PCDATA)>
```

B.2.2 Schema

```
<?xml version='1.0' encoding='UTF-8'?>
<xsd:schema
    xmlns:xsd='http://www.w3.org/2001/XMLSchema'
    targetNamespace='http://www.iana.org/assignments/sasl-mechanisms'
    xmlns='http://www.iana.org/assignments/sasl-mechanisms'
    elementFormDefault='qualified'>

  <xsd:element name='mechanisms'>
    <xsd:complexType>
      <xsd:sequence minOccurs='0' maxOccurs='unbounded'>
        <xsd:element ref='mechanism'/>
      </xsd:sequence>
    </xsd:complexType>
  </xsd:element>

  <xsd:element name='mechanism'>
    <xsd:complexType mixed='true'>
      <xsd:attribute name='name' type='xsd:string' use='optional'/>
    </xsd:complexType>
  </xsd:element>

  <xsd:element name='auth'>
    <xsd:complexType mixed='true'>
      <xsd:attribute name='name' type='xsd:string' use='optional'/>
    </xsd:complexType>
  </xsd:element>
  <xsd:element name='challenge' type='xsd:string'/>
  <xsd:element name='response' type='xsd:string'/>
  <xsd:element name='abort' type='xsd:string'/>
  <xsd:element name='success' type='xsd:string'/>
  <xsd:element name='failure' type='xsd:string'/>

</xsd:schema>
```

B.3 jabber:client namespace

Note: the formal definition for the 'jabber:server' namespace is identical to that for the 'jabber:client' namespace.

B.3.1 DTD

```
<?xml version='1.0' encoding='UTF-8'?>
<!ELEMENT message (( body? | subject? | thread? |
                     error? | (#PCDATA) )*)>

<!ATTLIST message
  to CDATA #IMPLIED
```

```
  from CDATA #IMPLIED
  id CDATA #IMPLIED
  type ( chat | groupchat | headline | error ) #IMPLIED
>

<!ELEMENT body (#PCDATA)>
<!ELEMENT subject (#PCDATA)>
<!ELEMENT thread (#PCDATA)>

<!ELEMENT presence (( show? | status? | priority? | error? )*)>

<!ATTLIST presence
  to CDATA #IMPLIED
  from CDATA #IMPLIED
  id CDATA #IMPLIED
  type ( subscribe | subscribed | unsubscribe |
         unsubscribed | unavailable | error ) #IMPLIED
>

<!ELEMENT show (#PCDATA)>
<!ELEMENT status (#PCDATA)>
<!ELEMENT priority (#PCDATA)>

<!ELEMENT iq ( error | (#PCDATA) )*>

<!ATTLIST iq
  to CDATA #IMPLIED
  from CDATA #IMPLIED
  id CDATA #IMPLIED
  type ( get | set | result | error ) #REQUIRED
>

<!ELEMENT error (#PCDATA)>
<!ATTLIST error code CDATA #REQUIRED>
```

B.3.2 Schema

```
<?xml version='1.0' encoding='UTF-8'?>
<xsd:schema
    xmlns:xsd='http://www.w3.org/2001/XMLSchema'
    targetNamespace='http://www.jabber.org/protocol'
    xmlns='http://www.jabber.org/protocol'
    elementFormDefault='qualified'>

  <xsd:element name='message'>
     <xsd:complexType mixed='true'>
        <xsd:choice>
```

```
            <xsd:element ref='body' minOccurs='0' maxOccurs='1'/>
            <xsd:element ref='subject' minOccurs='0' maxOccurs='1'/>
            <xsd:element ref='thread' minOccurs='0' maxOccurs='1'/>
            <xsd:element ref='error' minOccurs='0' maxOccurs='1'/>
            <xsd:any
                namespace='##other'
                minOccurs='0'
                maxOccurs='unbounded'/>
          </xsd:choice>
          <xsd:attribute name='to' type='xsd:string' use='optional'/>
          <xsd:attribute name='from' type='xsd:string' use='optional'/>
          <xsd:attribute name='id' type='xsd:string' use='optional'/>
          <xsd:attribute name='type' use='optional' default='normal'>
            <xsd:simpleType>
              <xsd:restriction base='xsd:NCName'>
                <xsd:enumeration value='normal'/>
                <xsd:enumeration value='chat'/>
                <xsd:enumeration value='groupchat'/>
                <xsd:enumeration value='headline'/>
                <xsd:enumeration value='error'/>
              </xsd:restriction>
            </xsd:simpleType>
          </xsd:attribute>
       </xsd:complexType>
    </xsd:element>

    <xsd:element name='body' type='xsd:string'/>

    <xsd:element name='subject' type='xsd:string'/>

    <xsd:element name='thread' type='xsd:string'/>

    <xsd:element name='presence'>
      <xsd:complexType>
        <xsd:choice>
          <xsd:element ref='show' minOccurs='0' maxOccurs='1'/>
          <xsd:element ref='status' minOccurs='0' maxOccurs='1'/>
          <xsd:element ref='priority' minOccurs='0' maxOccurs='1'/>
          <xsd:element ref='error' minOccurs='0' maxOccurs='1'/>
          <xsd:any
              namespace='##other'
              minOccurs='0'
              maxOccurs='unbounded'/>
        </xsd:choice>
        <xsd:attribute name='to' type='xsd:string' use='optional'/>
        <xsd:attribute name='from' type='xsd:string' use='optional'/>
        <xsd:attribute name='id' type='xsd:string' use='optional'/>
        <xsd:attribute name='type' use='optional'>
```

```
          <xsd:simpleType>
            <xsd:restriction base='xsd:string'>
              <xsd:enumeration value='unavailable'/>
              <xsd:enumeration value='subscribe'/>
              <xsd:enumeration value='subscribed'/>
              <xsd:enumeration value='unsubscribe'/>
              <xsd:enumeration value='unsubscribed'/>
              <xsd:enumeration value='error'/>
            </xsd:restriction>
          </xsd:simpleType>
        </xsd:attribute>
      </xsd:complexType>
    </xsd:element>

    <xsd:element name='show'>
      <xsd:simpleType>
        <xsd:restriction base='xsd:string'>
          <xsd:enumeration value='away'/>
          <xsd:enumeration value='chat'/>
          <xsd:enumeration value='xa'/>
          <xsd:enumeration value='dnd'/>
        </xsd:restriction>
      </xsd:simpleType>
    </xsd:element>

    <xsd:element name='status' type='xsd:string'/>

    <xsd:element name='priority' type='xsd:nonNegativeInteger'/>

    <xsd:element name='iq'>
      <xsd:complexType mixed='true'>
        <xsd:choice>
          <xsd:element ref='error' minOccurs='0' maxOccurs='1'/>
          <xsd:any
              namespace='##other'
              minOccurs='0'
              maxOccurs='unbounded'/>
        </xsd:choice>
        <xsd:attribute name='to' type='xsd:string' use='optional'/>
        <xsd:attribute name='from' type='xsd:string' use='optional'/>
        <xsd:attribute name='id' type='xsd:string' use='optional'/>
        <xsd:attribute name='type' use='required'>
          <xsd:simpleType>
            <xsd:restriction base='xsd:string'>
              <xsd:enumeration value='get'/>
              <xsd:enumeration value='set'/>
              <xsd:enumeration value='result'/>
              <xsd:enumeration value='error'/>
```

```
            </xsd:restriction>
          </xsd:simpleType>
        </xsd:attribute>
      </xsd:complexType>
    </xsd:element>

    <xsd:element name='error'>
      <xsd:complexType>
        <xsd:attribute
            name='code'
            type='xsd:nonNegativeInteger'
            use='required'/>
      </xsd:complexType>
    </xsd:element>

  </xsd:schema>
```

Appendix C. OpenPGP Usage

This section is non-normative. It describes an end-to-end encryption and signing method currently in use within the Jabber community. It is not recommended as a complete solution for encrypting streams or for guaranteeing the privacy of messages or presence. When this method is used, replay attacks are possible on presence chunks and also on messages for which the recipient is not mentioned in the message body. Key exchange may rely on the web of trust model used on the OpenPGP keys network. There is no method to check a fingerprint or ownership of a key other than checking the user IDs on a key.

All operations described herein may be completed using standard OpenPGP software. All program output is US-ASCII armored output with the headers removed, which allows for straightforward encapsulation of the program output directly in XML chunks. It is assumed that all keys are exchanged using OpenPGP key servers; for example, the key of another user may be retrieved automatically when a signed presence chunk is received from that user.

C.1 Signing Presence

Signing enables a sender to verify that they sent a certain block of text. As applied within the Jabber community, the <status/> child of a presence chunk is signed and sent as extended presence information in the 'jabber:x:signed' namespace. Because signing requires a block of text, a signed presence chunk MUST contain a <status/> child element that is non-empty (i.e., contains text.

When signing presence, the sender MUST use the private key which is the same KeyID as the one they wish to use for encrypted messages. This is because there is no feature negotiation related to message encryption; the only indicator that another user encrypts is or her messages is that one receives signed presence chunks from that user.

As shown in the following example, the only presence information that is signed is the CDATA of the <status> element.

```
<presence
    from='romeo@montague.net/orchard'
    to='juliet@capulet.com/balcony'>
  <status>Online</status>
  <x xmlns='jabber:x:signed'>
    iQA/AwUBOjU5dnol3d88qZ77EQI2JACfRngLJ045brNnaCX78ykKNUZaTIoAoPHI
    2uJxPMGR73EBIvEpcv0LRSy+
    =45f8
  </x>
</presence>
```

C.2 Encrypting Messages

Encryption enables the sender to encrypt a message sent to a specific recipient. This is accomplished by sending the encrypted form of the CDATA from the <body/> child in second child that is scoped by the 'jabber:x:encrypted' namespace. Because a block of text is necessary, the message chunk MUST contain a <body/> child element that is non-empty (i.e., that contains some CDATA text). It is considered polite to include a message <body/> informing the recipient that the message is encrypted. The public key used for message encryption should match the KeyID used for signing presence. The actual data that is encrypted is what would be the CDATA of the <body> element if the message were not encrypted.

```
<message
    from='juliet@capulet.com/balcony'
    to='romeo@montague.net/orchard'>
  <body>This message is encrypted.</body>
  <x xmlns='jabber:x:encrypted'>
    qANQR1DBwU4DX7jmYZnncmUQB/9KuKBddzQH+tZ1ZywKK0yHKnq57kWq+RFtQdCJ
    WpdWpR0uQsuJe7+vh3NWn59/gTc5MDlX8dS9p0ovStmNcyLhxVgmqS8ZKhsblVeu
    IpQ0JgavABqibJolc3BKrVtVV1igKiX/N7Pi8RtY1K18toaMDhdEfhBRzO/XB0+P
    AQhYlRjNacGcslkhXqNjK5Va4tuOAPy2n1Q8UUrHbUd0g+xJ9Bm0G0LZXyvCWyKH
    kuNEHFQiLuCY6Iv0myq6iX6tjuHehZlFSh80b5BVV9tNLwNR5Eqz1klxMhoghJOA
    w7R61cCPt8KSd8Vcl8K+StqOMZ5wkhosVjUqvEu8uJ9RupdpB/4m9E3gOQZCBsmq
    OsX4/jJhn2wIsfYYWdqkbNKnuYoKCnwrlmn6I+wX72p0R8tTv8peNCwK9bEtL/XS
    mhn4bCxoUkCITv3k8a+Jdvbov9ucduKSFuCBq4/l0fpHmPhHQjkFofxmaWJveFfF
    619NXyYyCfoLTmWk2AaTHVCjtKdf1WmwcTa0vFfk8BuFHkdah6kJJiJ7w/yNwa/E
    O6CMymuZTr/LpcKKWrWCt+SErxqmq8ekPI8h7oNwMxZBYAa7OJ1rXWKNgL9pDtNI
    824Mf0mXj7q5N1eMHvX1QEoKLAda/Ae3TTEevOyeUK1DEgvxfM2KRZ11RzU+XtIE
    My/bJk7EycAw8P/QKyeNlO1fxP58VEd6Gb8NCPqKOYn/LKh1O+c20ZNVEPFM4bNV
    XA4hB4UtFF7Ao8kpdlrUqdKyw4lEtnmdemYQ6+iIIVPEarWl9PxOMY90KAnZrSAq
    bt9uRY/1rPgelRaWblMKvxgpRO8++Y8VjdEyGgMOXxOiE851Ve72ftGzkSxDH8mW
    TgY3pf2aATmBp3lagQ1COkGS/xupovT5AQPA3RzbCxDvc6s6eGYKmVVQVj5vmSj1
    WULad5MB9KT1DzCm6FOSy063nWGBYYMWiejRvGLpo1j4eAnj0qOt7rTWmgv3RkYF
    Oin0vDOhW7aC
    =CvnG
  </x>
</message>
```

Full Copyright Statement

Acknowledgement

Funding for the RFC Editor function is currently provided by the Internet Society.