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

P. Saint-Andre
J. Miller
Jabber Software Foundation

February 03, 2003

XMPP Core
draft-ietf-xmpp-core-02

Status of this Memo

Copyright Notice

Abstract

This document describes the core features of the eXtensible Messaging and Presence Protocol (XMPP), a protocol for streaming XML in near-real-time that is used mainly for the purpose of instant messaging and presence 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. The protocol was developed originally within the Jabber community starting in 1998, and since 2001 has continued to evolve under the auspices of the Jabber Software Foundation and now the XMPP WG. 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 tens of thousands of domains to a user base that is estimated at over five million end users. The current document defines the core features 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 Terminology

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. The preferred forum is the <xmppwg@jabber.org> mailing list, for which 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.

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

```
C1 -  S1 - S2 - C3
     /  \
C2 -     G1 = FN1 = FC1
```

The symbols are as follows:

- C1, C2, C3 -- XMPP clients
- S1, S2 -- XMPP servers
- G1 -- A gateway that translates between XMPP and the protocol(s) used on a foreign (non-XMPP) messaging network
- FN1 -- A foreign messaging network
- FC1 -- A client on a foreign messaging network

2.2 Server

A server 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 clients and other servers) and to route appropriately-addressed XML data "stanzas" among such entities over XML streams. Most XMPP-compliant servers also assume responsibility for the storage of data that is used by clients (e.g., the contact list for each IM user); in this case, the XML data is processed directly by the server itself on behalf of the client and is not routed to another entity. Compliant server implementations MUST ensure in-order processing of XML stanzas received from connected clients, servers, and services.

2.3 Client

Most clients connect directly to a server over a TCP socket and use XMPP to take full advantage of the functionality provided by a server and any 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 server on behalf of each authorized client, with each resource connecting over a discrete TCP socket and differentiated by the resource identifier of a JID (Section 3) (e.g., user@domain/home vs. user@domain/work). The port assigned by the IANA [6] for connections between a Jabber client and a Jabber server is 5222. For further details about client-to-server communications expressly for the purpose of instant messaging and presence, refer to XMPP IM [2].

2.4 Gateway

A gateway is a special-purpose server-side 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 servers, and between gateways and the foreign messaging system, are not defined in this document.

2.5 Network

Because each server is identified by a network address (typically a DNS hostname) and because server-to-server communications are a straightforward extension of the client-to-server protocol, in practice the system consists of a network of servers that inter-communicate. Thus user-a@domain1 is able to exchange messages, presence, and other information with user-b@domain2. 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 servers is to open a TCP socket on the IANA-assigned port 5269 and to negotiate a connection using the Dialback Protocol (Section 5.2) 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 server to which the user connects, and the user's active session or connection (e.g., a specific client) in the form of user@domain/resource. However, node types other than clients are possible; for example, a specific chat room offered by a multi-user chat service could be addressed as room@service (where "room" is the name of the chat room and "service" is the hostname of the multi-user chat service) and a specific occupant of such a room could be addressed as room@service/nick (where "nick" is the occupant's room nickname).

3.2 Domain Identifier

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

The domain identifier for every server or service that will communicate over a network SHOULD resolve to a Fully Qualified Domain Name. A domain identifier MUST conform to RFC 952 [8] and RFC 1123 [9]. A domain identifier MUST be no more than 1023 bytes in length, and is subject to comparison in accordance with the rules defined in nameprep [10] profile of stringprep [11].

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 server or gateway (e.g., a client), although it can also represent other kinds of entities (e.g., a multi-user chat room

associated with a multi-user chat service). The entity represented by a node identifier is addressed within the context of a specific domain (e.g., user@domain).

A node identifier MUST be no more than 1023 bytes in length and MUST conform to the nodeprep [12] profile of stringprep [11].

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 the entity associated with a node identifier. An entity may maintain multiple resources simultaneously.

A resource identifier MUST be no more than 1023 bytes in length and MUST conform to the resourceprep [13] profile of stringprep [11].

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 stanzas". (Note: in this overview we use the example of communications between a client and server; however XML streams are more generalized and may be used for communications from server to server and from service to server as well.)

In order to connect to a server, a client must initiate an XML stream by sending a <stream> tag to the server, optionally preceded by a text declaration specifying the XML version supported and the character encoding. A compliant entity SHOULD 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 server SHOULD then reply with a second XML stream back to the client, again optionally preceded by a text declaration.

Within the context of an XML stream, a sender is able to send a discrete semantic unit of structured information to any recipient. This unit of structured information is a well-balanced XML stanza, such as a message, presence, or IQ stanza (a stanza of an XML document is said to be well-balanced if it matches production [43] content of the XML specification [1]). These stanzas exist at the direct child level of the root <stream/> element. The start of any XML stanza is unambiguously denoted by the element start tag at depth=1 (e.g., <presence>), and the end of any XML stanza is unambiguously denoted by the corresponding close tag at depth=1 (e.g., </presence>). Each XML stanza 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 client's request by sending a closing </stream> tag to the server (a session may also be closed by the server).

Thus a client's session with a server can be seen as two open-ended XML documents that are built up through the accumulation of the XML stanzas that are sent over the course of the session (one from the client to the server and one from the server to the client), and the root <stream/> element can be considered the document entity for those streams. In essence, then, an XML stream acts as an envelope for all the XML stanzas 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, predefined entities (as defined in Section 4.6 of the XML specification [1]), comments, or DTDs. Any such XML data SHOULD be ignored by a compliant implementation.

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 initiating 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.

- version -- The 'version' attribute MAY be used in the XML stream from the initiating entity to the receiving entity in order signal compliance with the protocol defined herein; this is done by setting the value of the attribute to "1.0". If the initiating entity includes the version attribute, the receiving entity MUST reciprocate by including the attribute in its response (if the receiving entity supports XMPP 1.0).

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
version	signals XMPP 1.0 support	signals XMPP 1.0 support

4.4 Namespace Declarations

The stream element MAY also contain namespace declarations as defined in the XML namespaces specification [14].

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. The default namespace declaration applies to the stream and all stanzas sent within a stream.

A stream namespace declaration (e.g., 'xmlns:stream') is REQUIRED in both XML streams. A compliant entity SHOULD 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".

XML streams function as containers for any XML stanzas 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 stanza over an XML stream. A compliant implementation MUST support the following two namespaces (for historical reasons, existing implementations MAY support only these two default namespaces):

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

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

The jabber:client and jabber:server namespaces are nearly identical but are used in different contexts (client-to-server communications for jabber:client and server-to-server communications for jabber:server). The only difference between the two is that the 'to' and 'from' attributes are OPTIONAL on stanzas sent within jabber:client, whereas they are REQUIRED on stanzas sent within jabber:server. If a compliant implementation accepts a stream that is scoped by the 'jabber:client' or 'jabber:server' namespace, it MUST support all three core stanza types (message, presence, and IQ) as described herein and defined in the DTD and schema.

4.5 Stream Features

The root stream element MAY contain a features child element (e.g., <stream:features/> if the stream namespace prefix is 'stream'). This is used to communicate generic stream-level capabilities including stream-level features that can be negotiated as the streams are set up. If the initiating entity sends a "version='1.0'" attribute in its initiating stream element, the receiving entity MUST send a features child element to the initiating entity if there are any capabilities that need to be advertised or features that can be negotiated for the stream. Currently this is used for SASL and TLS negotiation only, but it could be used for other negotiable features in the future (examples are shown under Stream Authentication (Section 5) below). If an entity does not understand or support some features, it SHOULD ignore them.

4.6 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 SHOULD be sent by a Jabber entity (usually a server rather than a client) if it perceives that a stream-level error has occurred. Examples include the sending of invalid XML, the shutdown

of a server, an internal server error such as the shutdown of a session manager, and an attempt by a client 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>
```

4.7 Simple Streams Example

The following is a stream-based session of a client on a server (where the "C" lines are sent from the client to the server, and the "S" lines are sent from the server to the client):

A basic session:

```
C: <?xml version='1.0'?>
   <stream:stream
       to='server'
       xmlns='jabber:client'
       xmlns:stream='http://etherx.jabber.org/streams'
       version='1.0'>
S: <?xml version='1.0'?>
   <stream:stream
       from='server'
       id='id_123456789'
       xmlns='jabber:client'
       xmlns:stream='http://etherx.jabber.org/streams'
       version='1.0'>
... authentication ...
C:   <message from='alex@graham-bell' to='watson@graham-bell'>
C:     <body>Watson come here, I want you!</body>
C:   </message>
S:   <message from='watson@graham-bell' to='alex@graham-bell'>
S:     <body>I'm on my way!</body>
S:   </message>
C: </stream:stream>
S: </stream:stream>
```

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

```
C: <?xml version='1.0'?>
   <stream:stream
       to='server'
       xmlns='jabber:client'
       xmlns:stream='http://etherx.jabber.org/streams'
       version='1.0'>
C:   <message from='alex@graham-bell' to='watson@graham-bell'>
C:     <body>Watson come here, I want you!</body>
C:   </message>
C: </stream:stream>

S: <?xml version='1.0'?>
   <stream:stream
       from='server'
       id='id_123456789'
       xmlns='jabber:client'
       xmlns:stream='http://etherx.jabber.org/streams'
       version='1.0'>
S:   <message from='watson@graham-bell' to='alex@graham-bell'>
S:     <body>I'm on my way!</body>
S:   </message>
S: </stream:stream>
```

A session gone bad:

```
C: <?xml version='1.0'?>
   <stream:stream
       to='server'
       xmlns='jabber:client'
       xmlns:stream='http://etherx.jabber.org/streams'
       version='1.0'>
S: <?xml version='1.0'?>
   <stream:stream
       from='server'
       id='id_123456789'
       xmlns='jabber:client'
       xmlns:stream='http://etherx.jabber.org/streams'
       version='1.0'>
C: <message><body>Bad XML, no closing body tag!</message>
S: <stream:error>Invalid XML</stream:error>
S: </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 user registers with a server or an administrator configures a server to trust another server), the preferred method for authenticating streams between the two entities uses an XMPP adaptation of the Simple Authentication and Security Layer (SASL) [15]. 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 servers is the server dialback protocol that is native to XMPP (no such ad-hoc method is defined between a client and a server). 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 RFC 2222 [15] (the namespace identifier for this protocol is 'http://www.iana.org/assignments/sasl-mechanisms'). If an entity (client or server) is capable of authenticating by means of SASL, it MUST include a 'version' attribute (set to a value of "1.0") within the opening <stream/> tag.

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

1. The client requests SASL authentication by including a 'version' attribute in the opening XML stream header sent to the server, with the value set to "1.0".

2. After sending an XML stream header in response, the server sends 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.

3. The client selects a mechanism by sending an <auth/> element to the server; this element MAY optionally contain character data if the mechanism supports or requires it.

4. If necessary, the server challenges the client by sending a

<challenge/> element to the client; this element MAY optionally contain character data.

5. The client responds to the challenge by sending a <response/> element to the server; this element MAY optionally contain character data.

6. If necessary, the server sends more challenges and the client sends more responses.

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

- The client aborts the handshake by sending an <abort/> element to the server.

- The server reports failure by sending a <failure/> element to the client.

- The server reports success by sending a <success/> element to the client; this element MAY optionally contain character data.

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

5.1.2 Client-Server Example

The following example shows the data flow for a client authenticating with a server using SASL.

Step 1: Client initiates stream to server:

```
<stream:stream
    xmlns='jabber:client'
    xmlns:stream='http://etherx.jabber.org/streams'
    to='capulet.com'
    version='1.0'>
```

Step 2: Server responds with a stream tag sent to the client:

```
<stream:stream
    xmlns='jabber:client'
    xmlns:stream='http://etherx.jabber.org/streams'
    id='12345678'
    version='1.0'>
```

Step 3: Server informs client of available authentication mechanisms:

```
<stream:features>
  <mechanisms xmlns='http://www.iana.org/assignments/sasl-mechanisms'>
    <mechanism>DIGEST-MD5</mechanism>
    <mechanism>PLAIN</mechanism>
  </mechanisms>
</stream:features>
```

Step 4: Client selects an authentication mechanism:

```
<auth
    xmlns='http://www.iana.org/assignments/sasl-mechanisms'
    mechanism='DIGEST-MD5'/>
```

Step 5: Server sends a base64-encoded challenge to the client:

```
<challenge xmlns='http://www.iana.org/assignments/sasl-mechanisms'>
    cmVhbG09ImNhdGFjbHlzbS5jeCIsbm9uY2U9Ik9BNk1HOXRFUUdtMmhoIi
    xxb3A9ImF1dGgiLGNoYXJzZXQ9dXRmLTgsYWxnb3JpdGhtPW1kNS1zZXNz
</challenge>
```

The decoded challenge is:

```
realm="cataclysm.cx",nonce="OA6MG9tEQGm2hh",\ qop="auth",charset=utf-
8,algorithm=md5-sess
```

Step 6: Client responds to the challenge:

```
<response xmlns='http://www.iana.org/assignments/sasl-mechanisms'>
    dXNlcm5hbWU9InJvYiIscmVhbG09ImNhdGFjbHlzbS5jeCIsbm9uY2U9Ik
    9BNk1HOXRFUUdtMmhoIixjbm9uY2U9Ik9BNk1IWGg2VnFUclJrIixuYz0w
    MDAwMDAwMSxxb3A9YXV0aCxkaWdlc3QtdXJpPSJqYWJiZXIvY2F0YWNseX
    NtLmN4IixyZXNwb25zZT1kMzg4ZGFkOTBkNGJiZDc2MGExNTIzMjFmMjE0
    M2FmNyxjaGFyc2V0PXV0Zi04
</response>
```

The decoded response is:

```
username="rob",realm="cataclysm.cx",nonce="OA6MG9tEQGm2hh",\
cnonce="OA6MHXh6VqTrRk",nc=00000001,qop=auth,\ digest-uri="jabber/
cataclysm.cx",\
response=d388dad90d4bbd760a152321f2143af7,charset=utf-8
```

Step 7: Server sends another challenge to the client:

```
<challenge xmlns='http://www.iana.org/assignments/sasl-mechanisms'>
    cnNwYXV0aD1lYTQwZjYwMzM1YzQyN2I1NTI3Yjg0ZGJhYmNkZmZmZA==
</challenge>
```

The decoded challenge is:

```
rspauth=ea40f60335c427b5527b84dbabcdfffd
```

Step 8: Client responds to the challenge:

```
<response xmlns='http://www.iana.org/assignments/sasl-mechanisms'/>
```

Step 9: Server informs client of successful authentication:

```
<success xmlns='http://www.iana.org/assignments/sasl-mechanisms'/>
```

Step 9 (alt): Server informs client of failed authentication:

```
<failure xmlns='http://www.iana.org/assignments/sasl-mechanisms'/>
```

5.2 Dialback Authentication

XMPP includes a protocol-level method for verifying that a connection between two servers can be trusted (at least as much as the DNS can 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 stanzas. Dialback is not intended as a mechanism for securing or encrypting the streams between servers, only for helping to prevent the spoofing of a server and the sending of false data from it. Dialback is made possible by the existence of DNS, since one server can verify that another server which is connecting to it is authorized to represent a given server on the Jabber network. All DNS hostname resolutions MUST first resolve the hostname using an SRV [22] record of _jabber._tcp.server. 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 server and the random ID generated for the stream. Generating unique but verifiable keys is important to prevent common man-in-the-middle attacks and server

spoofing.

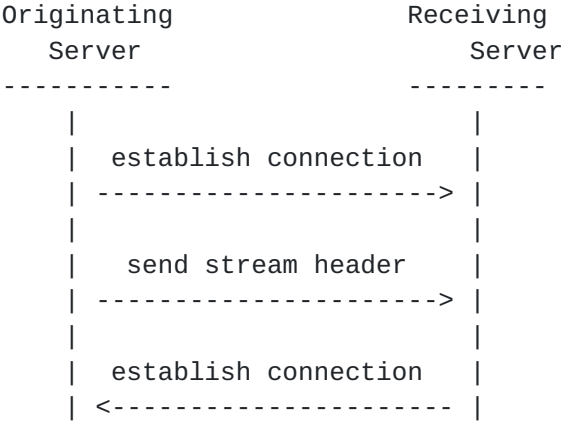
In the description that follows we use the following terminology:

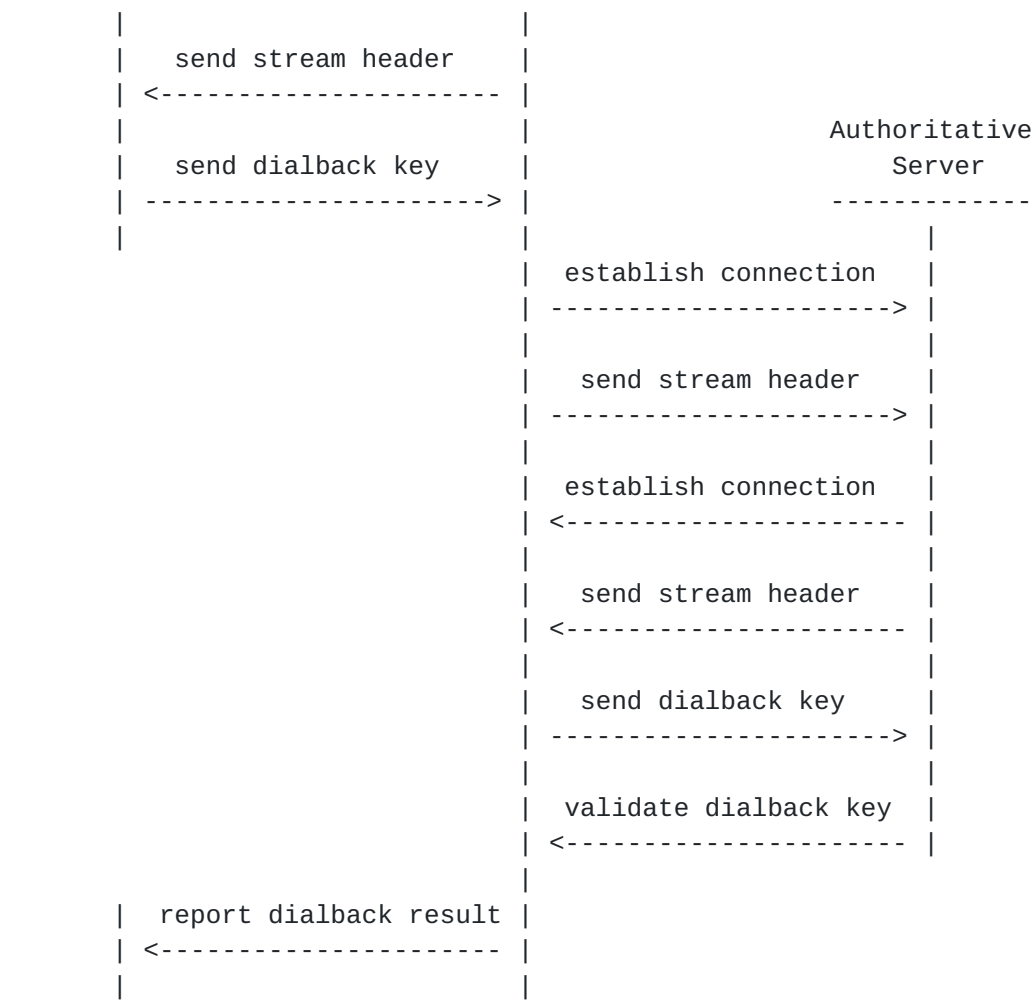
- Originating Server -- the server that is attempting to establish a connection between the two servers
- Receiving Server -- the server that is trying to authenticate that Originating Server represents the Jabber server which it claims to be
- Authoritative Server -- the server that is given when a DNS lookup is performed on the name that Originating Server initially gave; for basic environments this will be Originating Server, but it could be a separate machine in Originating Server's network

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

1. Originating Server establishes a connection to Receiving Server.
2. Originating Server sends a 'key' value over the connection to Receiving Server.
3. Receiving Server establishes a connection to Authoritative Server.
4. Receiving Server sends the same 'key' value to Authoritative Server.
5. Authoritative Server replies that key is valid or invalid.
6. Receiving Server tells Originating Server 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 servers is as follows:

1. Originating Server establishes TCP connection to Receiving Server

2. Originating Server sends a stream header to Receiving Server (the 'to' and 'from' attributes are NOT REQUIRED on the root stream element):

```
<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 Server that Originating Server supports dialback.

3. Receiving Server sends a stream header back to Originating Server (the 'to' and 'from' attributes are NOT REQUIRED on the root stream element):

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

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

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

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

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

6. Receiving Server sends Authoritative Server a stream header (the 'to' and 'from' attributes are NOT REQUIRED on the root stream element):

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

7. Authoritative Server sends Receiving Server a stream header:

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

8. Receiving Server sends Authoritative Server a stanza indicating it wants Authoritative Server to verify a key:

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

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

9. Authoritative Server sends a stanza back to Receiving Server verifying whether the key was valid or invalid:

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

or

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

10. Receiving Server informs Originating Server of the result:

```
<db:result
    from='Receiving Server'
    to='Originating Server'
    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 Originating Server and read by Receiving Server; before that, all data stanzas sent to Receiving Server SHOULD be dropped. As a final guard against domain spoofing, Receiving Server MUST verify that all XML stanzas received from Originating Server include a 'from' attribute and that the value of that attribute includes the validated domain. In addition, all XML stanzas MUST include a 'to' attribute.

6. Stream Encryption

6.1 Overview

XMPP includes a method for securing the stream from tampering and eavesdropping. This channel encryption method makes use of the Transport Layer Security (TLS) [16] protocol, along with a "STARTTLS" extension that is modelled on similar extensions for the IMAP [17], POP3 [18], and ACAP [19] protocols as described in RFC 2595 [20].

The namespace identifier for the STARTTLS extension is 'http://www.ietf.org/rfc/rfc2595.txt'. If an entity (client or server) is capable of using this extension, it MUST include the <starttls/> element in this namespace with the list of features that it sends in response to the opening stream tag that was used to initiate communications.

The following example shows the use of STARTTLS by a client to secure a session with a server, for which the steps involved are as follows:

1. The client opens a TCP connection and initiates the stream by sending the opening XML stream header to the server.

2. The server responds by opening a TCP connection and sending an XML stream header to the client.

3. The server offers the STARTTLS extension to the client by including it in the list of supported stream features.

4. The client issues the STARTTLS command to instruct the server that it wishes to begin a TLS negotiation to secure the stream.

5. The server closes the XML stream, but keeps the underlying TCP connection open. If the server is unable to prepare for the TLS negotiation for some reason, it returns an error.

6. The client begins a TLS negotiation according to RFC 2246 [16]. Upon completion of the negotiation, the client initiates a new stream by sending a new opening XML stream header to the server.

7. The server responds by sending an XML stream header to the client.

Once the stream is secured, the server MUST NOT offer the STARTTLS extension to the client.

6.2 Client-Server Example

The following example shows the data flow for a client securing a stream using STARTTLS.

Step 1: Client initiates stream to server:

```
<stream:stream
    xmlns='jabber:client'
    xmlns:stream='http://etherx.jabber.org/streams'
    to='capulet.com'
    version='1.0'>
```

Step 2: Server responds by sending a stream tag to the client:

```
<stream:stream
    xmlns='jabber:client'
    xmlns:stream='http://etherx.jabber.org/streams'
    id='12345678'
    version='1.0'>
```

Step 3: Server sends STARTTLS extensions to the client along with authentication mechanisms and any other stream features:

```
<stream:features>
  <starttls xmlns='http://www.ietf.org/rfc/rfc2595.txt'/>
  <mechanisms xmlns='http://www.iana.org/assignments/sasl-mechanisms'>
    <mechanism>DIGEST-MD5</mechanism>
    <mechanism>PLAIN</mechanism>
  </mechanisms>
</stream:features>
```

Step 4: Client sends the STARTTLS command to the server:

```
<starttls xmlns='http://www.ietf.org/rfc/rfc2595.txt'/>
```

Step 5: Server closes the stream:

```
</stream:stream>
```

Step 5 (alt): Server fails to prepare for the TLS negotiation:

```
<error xmlns='http://www.ietf.org/rfc/rfc2595.txt'/>
```

Step 6: Client begins TLS negotiation. When it has finished, it initiates a new stream to the server:

```
<stream:stream
    xmlns='jabber:client'
    xmlns:stream='http://etherx.jabber.org/streams'
    to='capulet.com'
    version='1.0'>
```

Step 7: Server responds by sending a stream tag to the client:

```
<stream:stream
    xmlns='jabber:client'
    xmlns:stream='http://etherx.jabber.org/streams'
    id='12345678'
    version='1.0'>
<stream:features>
  <mechanisms xmlns='http://www.iana.org/assignments/sasl-mechanisms'>
    <mechanism>DIGEST-MD5</mechanism>
    <mechanism>PLAIN</mechanism>
    <mechanism>EXTERNAL</mechanism>
  </mechanisms>
</stream:features>
```

6.3 Certificate-Based Authentication

If the client presents a valid client certificate during the TLS negotiation, the server MAY offer the SASL EXTERNAL mechanism to the client (see RFC 2222 [15]). If the client selects this mechanism for authentication, the authentication credentials shall be taken from the presented certificate.

7. XML Stanzas

7.1 Overview

Once the XML streams in each direction have been authenticated and (if desired) encrypted, XML stanzas can be sent over the streams. XML stanzas are the three core data elements for XMPP communications: <message/>, <presence/>, and <iq/>. These 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.

7.2 Common Attributes

Five attributes are common to message, presence, and IQ stanzas. These are defined below.

7.2.1 to

The 'to' attribute specifies the JID of the intended recipient for the stanza. In the 'jabber:client' namespace, a stanza SHOULD possess a 'to' attribute, although a stanza sent from a client to a server for handling by that server (e.g., presence sent to the server for broadcasting to other entities) MAY legitimately lack a 'to' attribute. In the 'jabber:server' namespace, a stanza MUST possess a 'to' attribute.

7.2.2 from

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

In the 'jabber:client' namespace, a client MUST NOT include a 'from' attribute on the stanzas it sends to a server; if a server receives a stanza from a client and the stanza possesses a 'from' attribute, it MUST ignore the value of the 'from' attribute. In addition, a server MUST stamp stanzas received from a client with the user@domain/resource (full JID) of the connected resource that generated the stanza.

In the 'jabber:server' namespace, a stanza MUST possess a 'from' attribute. In particular, a server MUST include a 'from' attribute on stanzas it routes to other servers. The domain identifier of the JID contained in the 'from' attribute MUST match the hostname of the server as communicated in the dialback negotiation (or a subdomain thereof).

7.2.3 id

The optional 'id' attribute MAY be used to track stanzas 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 stanza MAY return that 'id' in any replies, but is NOT REQUIRED to do so.

The value of the 'id' attribute is not intended to be unique -- globally, within a domain, or within a stream. It is generated by a sender only for internal tracking of information within the sending application.

7.2.4 type

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

7.2.5 xml:lang

Any message or presence stanza MAY possess an 'xml:lang' attribute specifying the default language of any CDATA sections of the stanza or its child elements. An IQ stanza SHOULD NOT possess an 'xml:lang' attribute, since it is merely a vessel for data in other namespaces and does not itself contain children that have CDATA. The value of the 'xml:lang' attribute MUST be an NMTOKEN and MUST conform to the format defined in RFC 3066 [21].

7.3 Message Stanzas

Message stanzas 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.

7.3.1 Types of Message

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

o 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 the meaning of these message types, refer to XMPP IM [2].

7.3.2 Children

If a message stanza 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 any 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 possess any attributes, with the exception of the 'xml:lang' attribute. Multiple instances of the <body/> element MAY be included but only if each instance possesses an 'xml:lang' attribute with a distinct language value.

- subject -- The subject of the message. The <subject/> element MUST NOT possess any attributes, with the exception of the 'xml:lang' attribute. Multiple instances of the <subject/> element MAY be included for the purpose of providing alternate versions of the same subject, but only if each instance possesses an 'xml:lang' attribute with a distinct language value.

- 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 (sometimes referred to as an "IM session") between two entities. If used, it MUST be unique to that conversation thread within the stream and MUST be consistent throughout that conversation. The use of the <thread/> element is optional and is not used to identify individual messages, only conversations. The method for generating thread IDs SHOULD be as follows: (1) concatenate the sender's full JID (user@host/resource) with the recipient's full JID; (2) concatenate these JID strings with a full ISO-8601 timestamp including year, month, day, hours, minutes, seconds, and UTC offset if appropriate in the

following format: yyyy-mm-dd-Thh:mm:ss-hh:mm; (3) hash the resulting string according to the SHA1 algorithm; (4) convert the hexidecimal SHA1 output to all lowercase. Only one <thread/> element MAY be included in a message stanza, and it MUST NOT possess any attributes. The <thread/> element MUST be treated as an opaque string by entities; no semantic meaning may be derived from it, and only exact, case-insensitve comparisons can be made against it.

If the message stanza 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), MAY possess an 'xml:lang' attribute, and MAY also contain PCDATA corresponding to a natural-language description of the error. An <error/> child MUST NOT be included if the stanza type is anything other than "error". An entity that receives a message stanza of type 'error' MUST NOT respond to the stanza by sending a further message stanza of type 'error'; this helps to prevent looping.

As described under extended namespaces (Section 7.6), a message stanza MAY also contain any properly-namespaced child element (other than the core data elements, stream elements, or defined children thereof).

7.4 Presence Stanzas

Presence stanzas 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 optional user-defined descriptive tex and optional user-defined descriptive textt) and to communicate that status to other entities. They are also used to negotiate and manage subscriptions to the presence of other entities.

7.4.1 Types of Presence

The 'type' attribute of a presence stanza is optional. A presence stanza that does not have a 'type' attribute is used to signal to the server 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 stanza. 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. In general SHOULD NOT be sent by a client.

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

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

7.4.2 Children

If a presence stanza possesses no 'type' attribute, it MAY contain any of the following child elements (note that the <status/> child MAY be sent in a presence stanza of type "unavailable" or, for historical reasons, "subscribe"):

- show -- Describes the availability status of an entity or specific resource. Only one <show/> element MAY be included in a presence stanza, and it MUST NOT possess any attributes. The value SHOULD be one of the following (values other than these four MAY be ignored; additional availability types could be defined through a properly-namespaced child element of the presence stanza):

  * 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"). The <status/> element MUST NOT possess any attributes, with the exception of the 'xml:lang' attribute. Multiple instances of the <status/> element MAY be included but only if each instance possesses an 'xml:lang' attribute with a distinct language value.

- o priority -- An optional element specifying the priority level of the connected resource. The value may be any integer between -128 to 127. Only one <priority/> element MAY be included in a presence stanza, and it MUST NOT possess any attributes.

If the presence stanza 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 contain PCDATA corresponding to a natural-language description of the error. An <error/> child MUST NOT be included if the stanza type is anything other than "error". An entity that receives a presence stanza of type 'error' MUST NOT respond to the stanza by sending a further presence stanza of type 'error'; this helps to prevent looping.

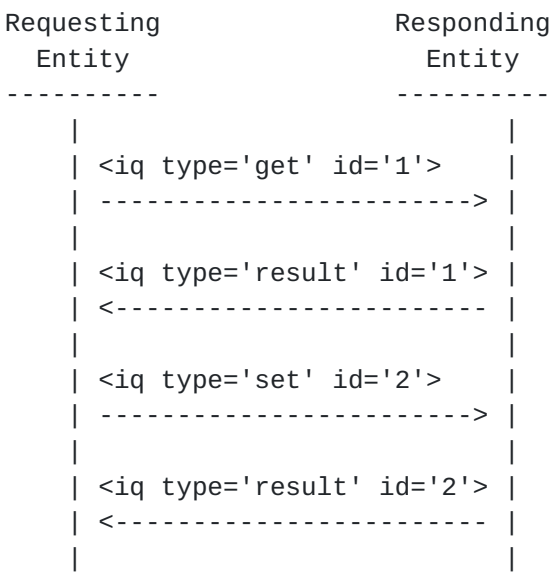
As described under extended namespaces (Section 7.6), a presence stanza MAY also contain any properly-namespaced child element (other than the core data elements, stream elements, or defined children thereof).

7.5 IQ Stanzas

7.5.1 Overview

Info/Query, or IQ, is a request-response mechanism. Just as HTTP is a request-response medium, so IQ stanzas 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, and the interaction is tracked by the requesting entity through use of the 'id' attribute, which responding entities SHOULD return in any response.

Most IQ interactions follow a common pattern of structured data exchange such as get/result or set/result (although an error may be returned in response to a request if appropriate):



An entity that receives an IQ request of type 'get' or 'set' MUST reply with an IQ response of type 'result' or 'error' (which response SHOULD preserve the 'id' attribute of the request). An entity that receives a stanza of type 'result' or 'error' MUST NOT respond to the stanza by sending a further IQ response of type 'result' or 'error'; however, as shown above, the requesting entity MAY send another request (e.g., an IQ of type 'set' in order to provide required information discovered through a get/result pair).

7.5.2 Types of IQ

The 'type' attribute of an IQ stanza 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 stanza is a request for information.
- set -- The stanza provides required data, sets new values, or replaces existing values.
- result -- The stanza 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.

7.5.3 Children

An IQ stanza 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 7.6), an IQ stanza MAY contain any properly-namespaced child element (other than the core data elements, stream elements, or defined children thereof).

If the IQ stanza 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 contain PCDATA corresponding to a natural-language description of the error. An <error/> child MUST NOT be included if the stanza type is anything other than "error". An entity that receives an IQ stanza of type 'error' MUST NOT respond to the stanza by sending a further IQ stanza of type 'error'; this helps to prevent looping.

7.6 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 stanza 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). 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 the stanza is being routed on to another entity, ignore means "pass it on untouched"). If an entity receives an IQ stanza in a namespace it does not understand, the entity SHOULD return an IQ stanza of type "error" with an error element of code 400 (bad request). If an entity receives a message or presence stanza that contains XML data in an extended namespace it does not understand, the portion of the stanza that is in the unknown namespace SHOULD be ignored. If an entity receives a message stanza without a <body/> element but containing only a child element bound by a namespace it does not understand, it MUST ignore the entire stanza.

8. XML Usage within XMPP

8.1 Namespaces

XML Namespaces [14] 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. Mainly for historical reasons, the default namespace for XMPP data stanzas 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.

8.2 Validation

A server is not responsible for validating the XML elements forwarded to a client; an implementation MAY choose to provide only validated data elements but is NOT REQUIRED to do so. Clients 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. Validation of XML streams and stanzas is NOT REQUIRED or recommended, and DTDs and schemas are included herein for descriptive purposes only.

8.3 Character Encodings

Software implementing XML streams MUST support the UTF-8 (RFC 2279 [24]) and UTF-16 (RFC 2781 [25]) transformations of Universal Character Set (ISO/IEC 10646-1 [26]) characters. Software MUST NOT attempt to use any other encoding for transmitted data. The encodings of the transmitted and received 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 the XML specification [1]. For historical reasons, existing implementations MAY support UTF-8 only.

8.4 Inclusion of Text Declaration

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

9. IANA Considerations

The IANA registers "jabber-client" and "jabber-server" as service names associated with TCP ports 5222 and 5269 respectively.

10. Internationalization Considerations

Usage of the 'xml:lang' attribute is described above. If a client includes an 'xml:lang' attribute in a stanza, the server MUST NOT modify or delete it.

11. Security Considerations

11.1 Client-to-Server Communications

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

The IP address and method of access of clients MUST NOT be made available by a server, nor are any connections other than the original server connection required. This helps protect the client's server 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 the methods defined in End-to-End Object Encryption in XMPP [27].

11.2 Server-to-Server Communications

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

If two servers 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 servers 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 servers.

11.3 Minimum Security Mechanisms

Although service provisioning is a policy matter, at a minimum, all implementations MUST support the following mechanisms:

for authentication: the SASL DIGEST-MD5 mechanism

for confidentiality: TLS (using the TLS_RSA_WITH_3DES_EDE_CBC_SHA cipher)

for both: TLS (using the TLS_RSA_WITH_3DES_EDE_CBC_SHA cipher supporting client-side certificates)

11.4 Firewalls

Communications using XMPP occur over TCP sockets on port 5222 (client-to-server) or port 5269 (server-to-server), as registered with the IANA [6]. Use of these well-known ports allows administrators to easily enable or disable XMPP activity through existing and commonly-deployed firewalls.

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] Saint-Andre, P. and J. Miller, "XMPP Instant Messaging (draft-ietf-xmpp-im-02, work in progress)", February 2003.

[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] Hoffman, P. and M. Blanchet, "Nameprep: A Stringprep Profile for Internationalized Domain Names (draft-ietf-idn-nameprep-11, work in progress)", June 2002.

[11] Hoffman, P. and M. Blanchet, "Preparation of Internationalized Strings ("stringprep")", RFC 3454, December 2002.

[12] Saint-Andre, P. and J. Hildebrand, "Nodeprep: A Stringprep Profile for Node Identifiers in XMPP (draft-ietf-xmpp-nodeprep-00, work in progress)", February 2003.

[13] Saint-Andre, P. and J. Hildebrand, "Resourceprep: A Stringprep Profile for Resource Identifiers in XMPP (draft-ietf-xmpp-resourceprep-00, work in progress)", February 2003.

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

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

[16] Dierks, T., Allen, C., Treese, W., Karlton, P., Freier, A. and P. Kocher, "The TLS Protocol Version 1.0", RFC 2246, January 1999.

[17] Crispin, M., "Internet Message Access Protocol - Version 4rev1", RFC 2060, December 1996.

[18] Myers, J. and M. Rose, "Post Office Protocol - Version 3", STD 53, RFC 1939, May 1996.

[19] Newman, C. and J. Myers, "ACAP -- Application Configuration Access Protocol", RFC 2244, November 1997.

[20] Newman, C., "Using TLS with IMAP, POP3 and ACAP", RFC 2595, June 1999.

[21] Alvestrand, H., "Tags for the Identification of Languages", BCP 47, RFC 3066, January 2001.

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

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

[24] Yergeau, F., "UTF-8, a transformation format of ISO 10646", RFC 2279, January 1998.

[25] Hoffman, P. and F. Yergeau, "UTF-16, an encoding of ISO 10646", RFC 2781, February 2000.

[26] International Organization for Standardization, "Information Technology - Universal Multiple-octet coded Character Set (UCS) - Amendment 2: UCS Transformation Format 8 (UTF-8)", ISO Standard 10646-1 Addendum 2, October 1996.

[27] Saint-Andre, P. and J. Hildebrand, "End-to-End Object Encryption in XMPP (draft-ietf-xmpp-e2e-00, work in progress)", February 2003.

Authors' Addresses

Peter Saint-Andre
Jabber Software Foundation

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

Jeremie Miller
Jabber Software Foundation

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

Appendix A. Standard Error Codes

A standard error element is used for failed processing of XML stanzas. This element is a child of the failed stanza 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 clients that they have provided incorrect authorization information, e.g., an incorrect password or unknown username when attempting to authenticate with a service.

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

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

- 404 (Not Found) - Code 404 is used to inform a sender that no recipient was found matching the JID to which an XML stanza was sent, e.g., if a sender has attempted to send a message to a JID that does not exist. (Note: if the server 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 client attempts to set the time or version of a server.

- 406 (Not Acceptable) - Code 406 is used when an XML stanza is for some reason not acceptable to a server or other entity. This might be generated when, for example, a user attempts to register with a service 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 user 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 server or service encounters an unexpected condition which prevents it from handling an XML stanza from a sender, e.g., if an authentication request is not handled by a server 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 user attempts to register with a server that does not allow registration.

- 502 (Remote Server Error) - Code 502 is used when delivery of an XML stanza fails because of an inability to reach the intended remote server or service, e.g., because a remote server'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 server timeout, e.g., if an incorrect hostname is specified.

Appendix B. Formal Definitions

B.1 streams namespace

The namespace declaration for the root stream element is 'http://etherx.jabber.org/streams'.

B.1.1 DTD

```
<?xml version='1.0' encoding='UTF-8'?>
<!ELEMENT stream (#PCDATA | error?)*>
<!ATTLIST stream
  to        CDATA   #IMPLIED
  from      CDATA   #IMPLIED
  id        ID      #IMPLIED
  version   CDATA   #IMPLIED
>
<!ELEMENT error (#PCDATA)>
<!ELEMENT features (#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:server'
             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:ID' use='optional'/>
      <xsd:attribute name='version' type='xsd:decimal' use='optional'/>
    </xsd:complexType>
  </xsd:element>

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

</xsd:schema>
```

B.2 SASL namespace

The namespace declaration for SASL-related elements is 'http://www.iana.org/assignments/sasl-mechanisms'.

B.2.1 DTD

```
<?xml version='1.0' encoding='UTF-8'?>
<!ELEMENT mechanisms (mechanism)*>
<!ELEMENT mechanism (#PCDATA)>
<!ELEMENT auth (#PCDATA)>
<!ATTLIST auth mechanism 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:element name='auth'>
    <xsd:complexType mixed='true'>
      <xsd:attribute name='mechanism' 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

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        ID       #IMPLIED
  xml:lang  NMTOKEN  #IMPLIED
  type ( chat | groupchat | headline | error ) #IMPLIED
```

```
>

<!ELEMENT body (#PCDATA)>
<!ATTLIST body xml:lang NMTOKEN #IMPLIED>
<!ELEMENT subject (#PCDATA)>
<!ATTLIST subject xml:lang NMTOKEN #IMPLIED>
<!ELEMENT thread (#PCDATA)>

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

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

<!ELEMENT show (#PCDATA)>
<!ELEMENT status (#PCDATA)>
<!ATTLIST status xml:lang NMTOKEN #IMPLIED>
<!ELEMENT priority (#PCDATA)>

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

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

<!ELEMENT error (#PCDATA)>
<!ATTLIST error
  code      CDATA    #REQUIRED
  xml:lang  NMTOKEN  #IMPLIED
>
```

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 maxOccurs='unbounded'>
              <xsd:element ref='body' minOccurs='0' maxOccurs='unbounded'/>
              <xsd:element ref='subject' minOccurs='0' maxOccurs='unbounded'/>
              <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:ID' use='optional'/>
           <xsd:attribute name='xml:lang' type='xsd:NMTOKEN' use='optional'/>
           <xsd:attribute name='type' use='optional'>
             <xsd:simpleType>
               <xsd:restriction base='xsd:NCName'>
                 <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:complexType>
         <xsd:attribute name='xml:lang' type='xsd:NMTOKEN' use='optional'/>
       </xsd:complexType>
     </xsd:element>

     <xsd:element name='subject' type='xsd:string'>
       <xsd:complexType>
         <xsd:attribute name='xml:lang' type='xsd:NMTOKEN' use='optional'/>
       </xsd:complexType>
     </xsd:element>

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

     <xsd:element name='presence'>
       <xsd:complexType>
         <xsd:choice maxOccurs='unbounded'>
           <xsd:element ref='show' minOccurs='0' maxOccurs='1'/>
           <xsd:element ref='status' minOccurs='0' maxOccurs='unbounded'/>
```

```
          <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:ID' use='optional'/>
        <xsd:attribute name='xml:lang' type='xsd:NMTOKEN' use='optional'/>
        <xsd:attribute name='type' use='optional'>
          <xsd:simpleType>
            <xsd:restriction base='xsd:NCName'>
              <xsd:enumeration value='subscribe'/>
              <xsd:enumeration value='subscribed'/>
              <xsd:enumeration value='unsubscribe'/>
              <xsd:enumeration value='unsubscribed'/>
              <xsd:enumeration value='unavailable'/>
              <xsd:enumeration value='error'/>
            </xsd:restriction>
          </xsd:simpleType>
        </xsd:attribute>
      </xsd:complexType>
    </xsd:element>

    <xsd:element name='show'>
      <xsd:simpleType>
        <xsd:restriction base='xsd:NCName'>
          <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:complexType>
        <xsd:attribute name='xml:lang' type='xsd:NMTOKEN' use='optional'/>
      </xsd:complexType>
    </xsd:element>

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

    <xsd:element name='iq'>
      <xsd:complexType mixed='true'>
        <xsd:choice maxOccurs='unbounded'>
```

```
          <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:ID' use='optional'/>
        <xsd:attribute name='type' use='required'>
          <xsd:simpleType>
            <xsd:restriction base='xsd:NCName'>
              <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:attribute name='xml:lang' type='xsd:NMTOKEN' use='optional'/>
      </xsd:complexType>
    </xsd:element>

  </xsd:schema>
```

B.4 jabber:server namespace

B.4.1 DTD

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

  <!ATTLIST message
    to        CDATA    #REQUIRED
    from      CDATA    #REQUIRED
    id        ID       #IMPLIED
    xml:lang  NMTOKEN  #IMPLIED
```

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

<!ELEMENT body (#PCDATA)>
<!ATTLIST body xml:lang NMTOKEN #IMPLIED>
<!ELEMENT subject (#PCDATA)>
<!ATTLIST subject xml:lang NMTOKEN #IMPLIED>
<!ELEMENT thread (#PCDATA)>

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

<!ATTLIST presence
  to        CDATA    #REQUIRED
  from      CDATA    #REQUIRED
  id        ID       #IMPLIED
  xml:lang  NMTOKEN  #IMPLIED
  type ( subscribe | subscribed | unsubscribe |
         unsubscribed | unavailable | error ) #IMPLIED
>

<!ELEMENT show (#PCDATA)>
<!ELEMENT status (#PCDATA)>
<!ATTLIST status xml:lang NMTOKEN #IMPLIED>
<!ELEMENT priority (#PCDATA)>

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

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

<!ELEMENT error (#PCDATA)>
<!ATTLIST error
  code      CDATA    #REQUIRED
  xml:lang  NMTOKEN  #IMPLIED
>
```

B.4.2 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 maxOccurs='unbounded'>
              <xsd:element ref='body' minOccurs='0' maxOccurs='unbounded'/>
              <xsd:element ref='subject' minOccurs='0' maxOccurs='unbounded'/>
              <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='required'/>
           <xsd:attribute name='from' type='xsd:string' use='required'/>
           <xsd:attribute name='id' type='xsd:ID' use='optional'/>
           <xsd:attribute name='xml:lang' type='xsd:NMTOKEN' use='optional'/>
           <xsd:attribute name='type' use='optional'>
             <xsd:simpleType>
               <xsd:restriction base='xsd:NCName'>
                 <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:complexType>
         <xsd:attribute name='xml:lang' type='xsd:NMTOKEN' use='optional'/>
       </xsd:complexType>
     </xsd:element>

     <xsd:element name='subject' type='xsd:string'>
       <xsd:complexType>
         <xsd:attribute name='xml:lang' type='xsd:NMTOKEN' use='optional'/>
       </xsd:complexType>
     </xsd:element>

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

     <xsd:element name='presence'>
       <xsd:complexType>
         <xsd:choice maxOccurs='unbounded'>
```

```
          <xsd:element ref='show' minOccurs='0' maxOccurs='1'/>
          <xsd:element ref='status' minOccurs='0' maxOccurs='unbounded'/>
          <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='required'/>
        <xsd:attribute name='from' type='xsd:string' use='required'/>
        <xsd:attribute name='id' type='xsd:ID' use='optional'/>
        <xsd:attribute name='xml:lang' type='xsd:NMTOKEN' use='optional'/>
        <xsd:attribute name='type' use='optional'>
          <xsd:simpleType>
            <xsd:restriction base='xsd:NCName'>
              <xsd:enumeration value='subscribe'/>
              <xsd:enumeration value='subscribed'/>
              <xsd:enumeration value='unsubscribe'/>
              <xsd:enumeration value='unsubscribed'/>
              <xsd:enumeration value='unavailable'/>
              <xsd:enumeration value='error'/>
            </xsd:restriction>
          </xsd:simpleType>
        </xsd:attribute>
      </xsd:complexType>
    </xsd:element>

    <xsd:element name='show'>
      <xsd:simpleType>
        <xsd:restriction base='xsd:NCName'>
          <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:complexType>
        <xsd:attribute name='xml:lang' type='xsd:NMTOKEN' use='optional'/>
      </xsd:complexType>
    </xsd:element>

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

    <xsd:element name='iq'>
```

```
      <xsd:complexType mixed='true'>
        <xsd:choice maxOccurs='unbounded'>
          <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='required'/>
        <xsd:attribute name='from' type='xsd:string' use='required'/>
        <xsd:attribute name='id' type='xsd:ID' use='optional'/>
        <xsd:attribute name='type' use='required'>
          <xsd:simpleType>
            <xsd:restriction base='xsd:NCName'>
              <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:attribute name='xml:lang' type='xsd:NMTOKEN' use='optional'/>
      </xsd:complexType>
    </xsd:element>

  </xsd:schema>
```

Appendix C. Revision History

Note to RFC editor: please remove this entire appendix, and the corresponding entries in the table of contents, prior to publication.

C.1 Changes from draft-ietf-xmpp-core-01

- Updated the addressing restrictions per list discussion and added references to the new nodeprep and resourceprep profiles.
- Corrected error in Stream Authentication regarding version='1.0' flag.
- Made numerous small editorial changes.

C.2 Changes from draft-ietf-xmpp-core-00

- Added information about TLS from list discussion.
- Clarified meaning of "ignore" based on list discussion.
- Clarified information about Universal Character Set data and character encodings.
- Provided base64-decoded information for examples.
- Fixed several errors in the schemas.
- Made numerous small editorial fixes.

C.3 Changes from draft-miller-xmpp-core-02

- Brought Streams Authentication section into line with discussion on list and at IETF 55 meeting.
- Added information about the optional 'xml:lang' attribute per discussion on list and at IETF 55 meeting.
- Specified that validation is neither required nor recommended, and that the formal definitions (DTDs and schemas) are included for descriptive purposes only.
- Specified that the response to an IQ stanza of type 'get' or 'set' must be an IQ stanza of type 'result' or 'error'.
- Specified that compliant server implementations must process

stanzas in order.

- Specified that for historical reasons some server implementations may accept 'stream:' as the only valid namespace prefix on the root stream element.

- Clarified the difference between 'jabber:client' and 'jabber:server' namespaces, namely, that 'to' and 'from' attributes are required on all stanzas in the latter but not the former.

- Fixed typo in Step 9 of the dialback protocol (changed db:result to db:verify).

- Removed references to TLS pending list discussion.

- Removed the non-normative appendix on OpenPGP usage pending its inclusion in a separate I-D.

- Simplified the architecture diagram, removed most references to services, and removed references to the 'jabber:component:*' namespaces.

- Noted that XMPP activity respects firewall administration policies.

- Further specified the scope and uniqueness of the 'id' attribute in all stanza types and the <thread/> element in message stanzas.

- Nomenclature changes: (1) from "chunks" to "stanzas"; (2) from "host" to "server" and from "node" to "client" (except with regard to definition of the addressing scheme).

Full Copyright Statement

Acknowledgement

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