Network Working Group
Internet-Draft
Expires: October 20, 2003

P. Saint-Andre
J. Miller
Jabber Software Foundation

April 21, 2003

XMPP Core
draft-ietf-xmpp-core-10

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 elements in order to exchange messages and presence information in close to real time. XMPP is used mainly for the purpose of building instant messaging (IM) and presence applications, such as the servers and clients 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, presence, and request-response services. The basic syntax and semantics were developed originally within the Jabber open-source community, mainly in 1999. In 2002, the XMPP WG was chartered with developing an adaptation of the Jabber protocol that would be suitable as an IETF instant messaging and presence technology. As a result of work by the XMPP WG, the current document defines the core features of XMPP; XMPP IM [22] defines the extensions required to provide the instant messaging (IM) and presence functionality defined in RFC 2779 [2].

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 [3].

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 [4] 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, servers, and other entities) 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., contact lists for users of XMPP-based IM applications); 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 between any two entities.

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. Although there is no necessary coupling of an XML stream to a TCP socket (e.g., a client COULD connect via HTTP polling or some other mechanism), this specification defines a binding for XMPP to TCP only. 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 registered with the IANA [5] for connections between a Jabber client and a Jabber server is 5222.

2.4 Gateway

A gateway is a special-purpose server-side service whose primary function is to translate XMPP into the protocol used by a foreign (non-XMPP) messaging system, as well as to translate the return data back into XMPP. Examples are gateways to SIMPLE, Internet Relay Chat (IRC), Short Message Service (SMS), SMTP, and legacy instant messaging networks such as AIM, ICQ, MSN Messenger, and Yahoo! Instant Messenger. 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. Upon opening a TCP socket on the IANA-registered port 5269, there are two methods for negotiating a connection between any two servers: primarily SASL authentication (Section 6.1) and secondarily server dialback (Section 6.2).

3. Addressing Scheme

3.1 Overview

An entity is anything that can be considered a network endpoint (i.e., an ID on the network) and that can communicate using XMPP. All such entities are uniquely addressable in a form that is consistent with RFC 2396 [23]. 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). Many other JID types are possible (e.g., <domain/resource> could be a server-side script or service).

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 [6] and RFC 1123 [7]. A domain identifier MUST be no more than 1023 bytes in length and MUST conform to the nameprep [8] profile of stringprep [9].

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 (i.e., 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; within IM applications of XMPP this address is called a "bare JID" and is of the form <user@domain>.

A node identifier MUST be no more than 1023 bytes in length and MUST conform to the nodeprep [10] profile of stringprep [9].

3.4 Resource Identifier

The resource identifier is an optional tertiary identifier, which may modify either a "user@domain" or mere "domain" address. It usually 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. A resource identifier is typically defined by a client implementation and is opaque to both servers and other clients. 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 [11] profile of stringprep [9].

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 XML stanzas. The terms may be defined as follows:

Definition of XML stream: An XML stream is a container for the exchange of XML elements between any two entities over a network. An XML stream is negotiated from an initiating entity (usually a client or server) to a receiving entity (usually a server), normally over a TCP socket, and corresponds to the initiating entity's "session" with the receiving entity. The start of the XML stream is denoted unambiguously by an opening XML <stream> tag with appropriate attributes and namespace declarations, and the end of the XML stream is denoted unambiguously be a closing XML </stream> tag. An XML stream is unidirectional; in order to enable bidirectional information exchange, the initiating entity and receiving entity must negotiate one stream in each direction, normally over the same TCP connection.

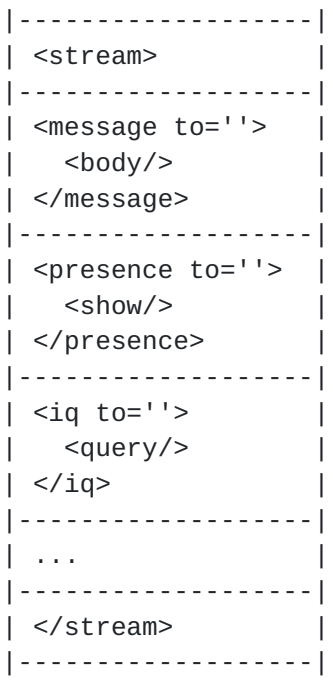
Definition of XML stanza: An XML stanza is a discrete semantic unit of structured information that is sent from one entity to another over an XML stream. An XML stanza exists at the direct child level of the root <stream/> element and is said to be well-balanced if it matches production [43] content of the XML specification [1]). The start of any XML stanza is denoted unambiguously by the element start tag at depth=1 (e.g., <presence>), and the end of any XML stanza is denoted unambiguously by the corresponding close tag at depth=1 (e.g., </presence>). An XML stanza MAY contain child elements (with accompanying attributes, elements, and CDATA) as necessary in order to convey the desired information.

Consider the example of a client's session with a server. In order to connect to a server, a client must initiate an XML stream by sending an opening <stream> tag to the server, optionally preceded by a text declaration specifying the XML version supported and the character encoding. The server SHOULD then reply with a second XML stream back to the client, again optionally preceded by a text declaration. Once the client has authenticated with the server (see Section 6), the client MAY send an unlimited number of XML stanzas over the stream to any recipient on the network. When the client desires to close the stream, it simply sends a closing </stream> tag to the server (alternatively, the session may be closed by the server), after which both the client and server SHOULD close the

underlying TCP connection as well.

Those who are accustomed to thinking of XML in a document-centric manner may wish to view a client's session with a server as consisting of two open-ended XML documents: one from the client to the server and one from the server to the client. From this perspective, the root <stream/> element can be considered the document entity for each "document", and the two "documents" are built up through the accumulation of XML stanzas sent over the two XML streams. However, this perspective is a convenience only, and XMPP does not deal in documents but in XML streams and XML stanzas.

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:



4.2 Stream Attributes

The attributes of the stream element are as follows:

o to -- The 'to' attribute SHOULD be used only in the XML stream header from the initiating entity to the receiving entity, and MUST be set to the XMPP address of the receiving entity. There SHOULD be no 'to' attribute set in the XML stream header by which the receiving entity replies to the initiating entity; however, if a 'to' attribute is included, it SHOULD be silently ignored by the initiating entity.

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

- id -- The 'id' attribute SHOULD be used only in the XML stream header 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 header sent from the initiating entity to the receiving entity; however, if an 'id' attribute is included, it SHOULD be silently ignored by the receiving entity.

- version -- The 'version' attribute MAY be used in the XML stream header 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 and the receiving entity supports XMPP 1.0, the receiving entity MUST reciprocate by including the attribute in its response.

We can summarize these values as follows:

	initiating to receiving	receiving to initiating
to	hostname of receiver	silently ignored
from	silently ignored	hostname of receiver
id	silently ignored	session key
version	signals XMPP 1.0 support	signals XMPP 1.0 support

4.3 Namespace Declarations

The stream element MAY contain namespace declarations as defined in the XML namespaces specification [12].

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 the use of a <stream:stream/> element as the stream root. The name 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 define 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 (unless explicitly scoped by another namespace).

Since 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). At a minimum, a compliant implementation MUST support the following two namespaces (for historical reasons, some 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 schema.

4.4 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'" flag 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 (usage is defined under Stream Encryption (Section 5) and Stream Authentication (Section 6) below). If an entity does not understand or support some features, it SHOULD silently ignore them.

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 MUST be sent by a compliant entity (usually a server rather than a client) if it perceives that a stream-level error has occurred.

4.5.1 Rules

The following rules apply to stream-level errors:

- It is assumed that all stream-level errors are unrecoverable; therefore, if an error occurs at the level of the stream, the entity that detects the error MUST send a stream error to the other entity, send a closing </stream> tag, and close the underlying TCP connection.

- If the error occurs while the stream is being set up, the receiving entity MUST still send the opening and closing stream tags and include the error element as a child of the stream element. In this case, if the initiating entity provides an unknown host in the 'to' attribute (or provides no 'to' attribute at all), the server SHOULD provide the server's authoritative hostname in the 'from' attribute of the stream header sent before termination.

4.5.2 Syntax

The syntax for stream errors is as follows:

```
<stream:error class='error-class'>
  <condition xmlns='urn:ietf:params:xml:ns:xmpp-streams'>
    <descriptive-element-name/>
  </condition>
</stream:error>
```

The value of the 'class' attribute must be one of the following:

- address -- the condition relates to the JID or domain to which the stream was addressed

- format -- the condition relates to XML format or structure

- redirect -- the condition relates to a host redirection

- server -- the condition relates to the internal state of the

server

The <condition/> element MUST contain a child element that specifies a particular stream-level error condition, as defined in the next section. (Note: the XML namespace name 'urn:ietf:params:xml:ns:xmpp-streams' that scopes the <condition/> element adheres to the format defined in The IETF XML Registry [24].)

4.5.3 Conditions

The following stream-level error conditions are defined:

- <host-gone/> -- the value of the 'to' attribute provided by the initiating entity in the stream header corresponds to a hostname that is no longer hosted by the server; the associated class is "address".

- <host-unknown/> -- the value of the 'to' attribute provided by the initiating entity in the stream header does not correspond to a hostname that is hosted by the server; the associated class is "address".

- <internal-server-error/> -- the server has experienced a misconfiguration or an otherwise-undefined internal server error that prevents it from servicing the stream; the associated class is "server".

- <invalid-id/> -- the stream ID or dialback ID is invalid or does not match an ID previously provided; the associated class is "format".

- <invalid-namespace/> -- the stream namespace name is something other than "http://etherx.jabber.org/streams" or the dialback namespace name is something other than "jabber:server:dialback"; the associated class is "format".

- <nonmatching-hosts/> -- the hostname provided in a 'from' address does not match the hostname (or any validated domain) negotiated via SASL or dialback; the associated class is "address".

- <not-authorized/> -- the entity does not possess sufficient privilegs to perform the desired action; the associated class is "access".

- <remote-connection-failed/> -- the server is unable to properly connect to a remote resource that is required for authentication or authorization; the associated class is "server".

- <resource-constraint/> -- the server is resource-contrained and is unable to service the stream; the associated class is "server".

- <see-other-host/> -- the server will not provide service to the initiating entity but is redirecting traffic to another host; this element SHOULD contain CDATA specifying the alternate hostname or IP address to which the initiating entity MAY attempt to connect; the associated class is "redirect".

- <system-shutdown/> -- the server is being shut down and all active streams are being closed; the associated class is "server".

- <unsupported-stanza-type/> -- the initiating entity has sent a first-level child of the stream that is not supported by the server; the associated class is "format".

- <unsupported-version/> -- the value of the 'version' attribute provided by the initiating entity in the stream header specifies a version of XMPP that is not supported by the server; this element MAY contain CDATA specifying the XMPP version(s) supported by the server; the associated class is "format".

- <xml-not-well-formed/> -- the initiating entity has sent XML that is not well-formed as defined by the XML specification [1]; the associated class is "format".

4.5.4 Extensibility

If desired, an XMPP application MAY provide custom error information; this MUST be contained in a properly-namespaced child of the <condition/> element (i.e., the namespace name MUST NOT be one of the namespace names defined herein).

4.6 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='shakespeare.lit'
       xmlns='jabber:client'
       xmlns:stream='http://etherx.jabber.org/streams'
       version='1.0'>
S: <?xml version='1.0'?>
   <stream:stream
       from='shakespeare.lit'
       id='id_123456789'
       xmlns='jabber:client'
       xmlns:stream='http://etherx.jabber.org/streams'
       version='1.0'>
... authentication ...
C:   <message from='juliet@shakespeare.lit'
              to='romeo@shakespeare.lit'>
C:     <body>Art thou not Romeo, and a Montague?</body>
C:   </message>
S:   <message from='romeo@shakespeare.lit'
              to='juliet@shakespeare.lit'>
S:     <body>Neither, fair saint, if either thee dislike.</body>
S:   </message>
C: </stream:stream>
S: </stream:stream>
```

A session gone bad:

```
C: <?xml version='1.0'?>
   <stream:stream
       to='shakespeare.lit'
       xmlns='jabber:client'
       xmlns:stream='http://etherx.jabber.org/streams'
       version='1.0'>
S: <?xml version='1.0'?>
   <stream:stream
       from='shakespeare.lit'
       id='id_123456789'
       xmlns='jabber:client'
       xmlns:stream='http://etherx.jabber.org/streams'
       version='1.0'>
... authentication ...
C: <message><body>Bad XML, no closing body tag!</message>
S: <stream:error class='client'>
     <condition xmlns='urn:ietf:params:xml:ns:xmpp-streams'>
       <xml-not-well-formed/>
     </condition>
   </stream:error>
S: </stream:stream>
```

5. Stream Encryption

5.1 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) [13] protocol, along with a "STARTTLS" extension that is modelled on similar extensions for the IMAP [25], POP3 [26], and ACAP [27] protocols as described in RFC 2595 [28]. The namespace identifier for the STARTTLS extension is 'urn:ietf:params:xml:ns:xmpp-tls'.

TLS SHOULD be used between any initiating entity and any receiving entity (e.g., a stream from a client to a server or from one server to another). An administrator of a given domain MAY require use of TLS for either or both client-to-server communications and server-to-server communications. Servers SHOULD use TLS betweeen two domains for the purpose of securing server-to-server communcations. When the remote domain is already known, the server can verify the credentials of the known domain by comparing known keys or certificates. When the remote domain is not recognized, it may still be possible to verify a certificate if it is signed by a common trusted authority. Even if there is no way to verify certificates (e.g., an unknown domain with a self-signed certificate, or a certificate signed by an unrecognized authority), if the servers choose to communicate despite the lack of verified credentials, TLS still SHOULD be used to provide encryption.

The following business rules apply:

1. An initiating entity that complies with this specification MUST include the "version='1.0'" flag in the initiating stream header.

2. When a receiving entity that complies with this specification receives an initiating stream header that includes the "version='1.0'" flag, after sending a stream header in reply it MUST also send a <starttls/> element scoped by the 'urn:ietf:params:xml:ns:xmpp-tls' namespace as well as the list of other stream features it supports.

3. If the initiating entity chooses to use TLS for stream encryption, TLS negotiation MUST be completed before proceeding to SASL negotiation.

4. The initiating entity MUST validate the certificate presented by the receiving entity:

    1. If the initiating entity has been configured with a set of

trusted roots, either a well-known public set or a manually configured Certificate Authority (e.g., an organization's own Certificate Authority), normal certificate validation processing is appropriate.

2. If the initiating entity has been configured with the receiving entity's public key or certificate, a simple comparison is appropriate.

If the above methods fail, the certificate MAY be presented to a user for approval; the user SHOULD be given the option to store the certificate and not ask again for at least some reasonable period of time.

5. If the TLS negotiation is successful, the receiving entity MUST discard any knowledge obtained from the initiating entity before TLS takes effect.

6. If the TLS negotiation is successful, the initiating entity MUST discard any knowledge obtained from the receiving entity before TLS takes effect.

7. If the TLS negotiation is successful, the receiving entity MUST NOT offer the STARTTLS extension to the initiating entity along with the other stream features that are offered when the stream is restarted.

8. If the TLS negotiation results in success, the initiating entity SHOULD continue with SASL negotiation.

9. If the TLS negotiation results in failure, the receiving entity MUST terminate both the XML stream and the underlying TCP connection.

5.2 Narrative

When an initiating entity secures a stream with a receiving entity, the steps involved are as follows:

1. The initiating entity opens a TCP connection and initiates the stream by sending the opening XML stream header to the receiving entity, including the "version='1.0'" flag.

2. The receiving entity responds by opening a TCP connection and sending an XML stream header to the initiating entity.

3. The receiving entity offers the STARTTLS extension to the

initiating entity by sending it along with the list of supported stream features.

4. The initiating entity issues the STARTTLS command to instruct the receiving entity that it wishes to begin a TLS negotiation to secure the stream.

5. The receiving entity MUST reply with either a <proceed/> element or a <failure/> element scoped by the 'urn:ietf:params:xml:ns:xmpp-tls' namespace, but keep the underlying TCP connection open.

6. The initiating entity begins a TLS negotiation in accordance with RFC 2246 [13]. Upon completion of the negotiation, the initiating entity initiates a new stream by sending a new opening XML stream header to the receiving entity.

7. The receiving entity responds by sending an XML stream header to the initiating entity along with the remaining available features (but NOT including the STARTTLS element).

5.3 Client-to-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 the STARTTLS extension to the client along with authentication mechanisms and any other stream features (if TLS is required for interaction with this server, the server SHOULD signal that fact by including a <required/> element as a child of the <starttls/> element):

```
<stream:features>
  <starttls xmlns='urn:ietf:params:xml:ns:xmpp-tls'>
    <required/>
  </starttls>
  <mechanisms xmlns='urn:ietf:params:xml:ns:xmpp-sasl'>
    <mechanism>DIGEST-MD5</mechanism>
    <mechanism>PLAIN</mechanism>
  </mechanisms>
</stream:features>
```

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

```
<starttls xmlns='urn:ietf:params:xml:ns:xmpp-tls'/>
```

Step 5: Server informs client to proceed:

```
<proceed xmlns='urn:ietf:params:xml:ns:xmpp-tls'/>
```

Step 5 (alt): Server informs client that TLS negotiation has failed and closes stream:

```
<failure xmlns='urn:ietf:params:xml:ns:xmpp-tls'/>
</stream:stream>
```

Step 6: Client and server complete TLS negotiation over the existing TCP connection.

Step 7: Client 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 8: Server responds by sending a stream header to the client along with any remaining negotiatiable stream features:

```
<stream:stream
    xmlns='jabber:client'
    xmlns:stream='http://etherx.jabber.org/streams'
    id='12345678'
    version='1.0'>
<stream:features>
  <mechanisms xmlns='urn:ietf:params:xml:ns:xmpp-sasl'>
    <mechanism>DIGEST-MD5</mechanism>
    <mechanism>PLAIN</mechanism>
    <mechanism>EXTERNAL</mechanism>
  </mechanisms>
</stream:features>
```

Step 9: Client SHOULD continue with stream authentication (Section 6).

5.4 Server-to-Server Example

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

Step 1: Server1 initiates stream to Server2:

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

Step 2: Server2 responds by sending a stream tag to Server1:

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

Step 3: Server2 sends the STARTTLS extension to Server1 along with authentication mechanisms and any other stream features (if TLS is required for interaction with Server2, it SHOULD signal that fact by including a <required/> element as a child of the <starttls/> element):

```
<stream:features>
  <starttls xmlns='urn:ietf:params:xml:ns:xmpp-tls'/>
    <required/>
  </starttls>
  <mechanisms xmlns='urn:ietf:params:xml:ns:xmpp-sasl'>
    <mechanism>DIGEST-MD5</mechanism>
    <mechanism>KERBEROS_V4</mechanism>
  </mechanisms>
</stream:features>
```

Step 4: Server1 sends the STARTTLS command to Server2:

```
<starttls xmlns='urn:ietf:params:xml:ns:xmpp-tls'/>
```

Step 5: Server2 informs Server1 to proceed:

```
<proceed xmlns='urn:ietf:params:xml:ns:xmpp-tls'/>
```

Step 5 (alt): Server2 informs Server1 that TLS negotiation has failed and closes stream:

```
<failure xmlns='urn:ietf:params:xml:ns:xmpp-tls'/>
</stream:stream>
```

Step 6: Server1 and Server2 complete TLS negotiation via TCP.

Step 7: Server1 initiates a new stream to Server2:

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

Step 8: Server2 responds by sending a stream header to Server1 along with any remaining negotiatiable stream features:

```
<stream:stream
    xmlns='jabber:server'
    xmlns:stream='http://etherx.jabber.org/streams'
    id='12345678'
    version='1.0'>
<stream:features>
  <mechanisms xmlns='urn:ietf:params:xml:ns:xmpp-sasl'>
    <mechanism>DIGEST-MD5</mechanism>
    <mechanism>KERBEROS_V4</mechanism>
    <mechanism>EXTERNAL</mechanism>
  </mechanisms>
</stream:features>
```

Step 9: Server1 SHOULD continue with stream authentication (Section 6).

6. Stream Authentication

XMPP includes two methods for enforcing authentication at the level of XML streams. The secure and preferred method for authenticating streams between two entities uses an XMPP adaptation of the Simple Authentication and Security Layer (SASL) [14]. If SASL negotiation is not possible, some level of trust MAY be established based on existing trust in DNS; the authentication method used in this case is the server dialback protocol that is native to XMPP (no such ad-hoc method is defined between a client and a server). If SASL is used for server-to-server authentication, the servers MUST NOT use dialback. For further information about the relative merits of these two methods, consult Security Considerations (Section 11).

Stream authentication is REQUIRED for all direct communications between two entities; if an entity sends a stanza to an unauthenticated stream, the receiving entity SHOULD silently drop the stanza and MUST NOT process it.

6.1 SASL Authentication

6.1.1 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 [14] (the XMPP-specific namespace identifier is 'urn:ietf:params:xml:ns:xmpp-sasl').

The following business rules apply:

1. If TLS is used for stream encryption, SASL MUST NOT be used for anything but stream authentication (i.e., a security layer MUST NOT be negotiated using SASL). Conversely, if a security layer is to be negotiated via SASL, TLS MUST NOT be used.

2. If the initiating entity is capable of authenticating via SASL, it it MUST include the "version='1.0'" flag in the initiating stream header.

3. If the receiving entity is capable of accepting authentications via SASL, it MUST send one or more authentication mechanisms within a <mechanisms/> element scoped by the 'urn:ietf:params:xml:ns:xmpp-sasl' namespace in response to the opening stream tag received from the initiating entity.

4. If the SASL negotiation involves negotiation of a security layer,

the receiving entity MUST discard any knowledge obtained from the initiating entity which was not obtained from the SASL negotiation itself.

5. If the SASL negotiation involves negotiation of a security layer, the initiating entity MUST discard any knowledge obtained from the receiving entity which was not obtained from the SASL negotiation itself.

The following syntax rules apply:

1. The initial challenge MUST include a realm, nonce, qop, charset, and algorithm.

2. The inital response for client-to-server negotiation MUST include a username, realm, nonce, cnonce, nc, qop, digest-uri, response, charset, and authzid.

3. The inital response for server-to-server negotiation MUST include a realm, nonce, cnonce, nc, qop, digest-uri, response, and charset.

4. The realm-value MUST be no more than 1023 bytes in length and MUST conform to the nameprep [8] profile of stringprep [9].

5. The username-value MUST be no more than 1023 bytes in length and MUST conform to the nodeprep [10] profile of stringprep [9].

6. The response-value MUST be computed in accordance with the relevant SASL mechanism as defined by the appropriate RFC (e.g., RFC 2831 [15] for digest authentication).

7. The resource identifier portion of the authzid-value MUST be no more than 1023 bytes in length and MUST conform to the resourceprep [11] profile of stringprep [9].

6.1.2 Narrative

When an initiating entity authenticates with a receiving entity, the steps involved are as follows:

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

2. After sending an XML stream header in response, the receiving entity sends a list of available SASL authentication mechanisms,

each of which is a <mechanism/> element included as a child within a <mechanisms/> container element scoped by the 'urn:ietf:params:xml:ns:xmpp-sasl' namespace that is sent as a child of a <features/> element in the streams namespace. If channel encryption must be established before a particular authentication mechanism may be used, the receiving entity MUST NOT provide that mechanism in the list of available SASL authentication methods. If the initiating entity presents a valid initiating entity certificate during TLS negotiation, the receiving entity MAY offer the SASL EXTERNAL mechanism to the initiating entity during stream authentication (see RFC 2222 [14]).

3. The initiating entity selects a mechanism by sending an <auth/> element scoped by the 'urn:ietf:params:xml:ns:xmpp-sasl' namespace to the receiving entity; this element MAY optionally contain character data (in SASL terminology the "initial response") if the mechanism supports or requires it. If the initiating entity selects the EXTERNAL mechanism for authentication, the authentication credentials shall be taken from the certificate presented during TLS negotiation.

4. If necessary, the receiving entity challenges the initiating entity by sending a <challenge/> element scoped by the 'urn:ietf:params:xml:ns:xmpp-sasl' namespace to the initiating entity; this element MAY optionally contain character data (which MUST be computed in accordance with the SASL mechanism chosen by the initiating entity).

5. The initiating entity responds to the challenge by sending a <response/> element scoped by the 'urn:ietf:params:xml:ns:xmpp-sasl' namespace to the receiving entity; this element MAY optionally contain character data (which MUST be computed in accordance with the SASL mechanism chosen by the initiating entity).

6. If necessary, the receiving entity sends more challenges and the initiating entity sends more responses.

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

1. The initiating entity aborts the handshake by sending an <abort/> element to the receiving entity.

2. The receiving entity reports failure of the handshake by sending a <failure/> element scoped by the 'urn:ietf:params:xml:ns:xmpp-sasl' namespace to the initiating entity. The particular cause

of failure SHOULD be communicated in an appropriate child element of the <failure/> element. The following conditions are defined:

* <authentication-mechanism-too-weak/>

* <invalid-realm/>

* <not-authorized/>

* <password-transition-required/>

* <temporary-authentication-failure/>

3. The receiving entity reports success of the handshake by sending a <success/> element scoped by the 'urn:ietf:params:xml:ns:xmpp-sasl' namespace to the initiating entity; this element MAY optionally contain character data (in SASL terminology "additional data with success").

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

6.1.3 SASL Definition

Section 4 of the SASL specification [14] requires that the following information be supplied by a protocol definition:

service name: "xmpp"

initiation sequence: After the initiating entity provides an opening XML stream header and the receiving entity replies in kind, the receiving entity provides a list of acceptable authentication methods. The initiating entity chooses one method from the list and sends it to the receiving entity as the value of the 'mechanism' attribute possessed by an <auth/> element, optionally including an initial response to avoid a round trip.

exchange sequence: Challenges and responses are carried through the exchange of <challenge/> elements from receiving entity to initiating entity and <response/> elements from initiating entity to receiving entity. The receiving entity reports failure by sending a <failure/> element and success by sending a <success/> element; the initiating entity aborts the exchange by sending an <abort/> element. (All of these elements are scoped by the 'urn:ietf:params:xml:ns:xmpp-sasl' namespace.)

security layer negotiation: If a security layer is negotiated, both sides consider the original stream closed and new <stream/>

headers are sent by both entities. The security layer takes effect immediately following the ">" character of the <response/> element for the client and immediately following the closing ">" character of the <succeed/> element for the server. (Both of these elements are scoped by the 'urn:ietf:params:xml:ns:xmpp-sasl' namespace.)

use of the authorization identity: The authorization identity is used by xmpp only in negotiation between a client and a server, and denotes the "full JID" (user@domain/resource) requested by the user or application associated with the client.

6.1.4 Client-to-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='domain'
    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'
    from='domain'
    version='1.0'>
```

Step 3: Server informs client of available authentication mechanisms:

```
<stream:features>
  <mechanisms xmlns='urn:ietf:params:xml:ns:xmpp-sasl'>
    <mechanism>DIGEST-MD5</mechanism>
    <mechanism>PLAIN</mechanism>
  </mechanisms>
</stream:features>
```

Step 4: Client selects an authentication mechanism:

```
<auth
    xmlns='urn:ietf:params:xml:ns:xmpp-sasl'
    mechanism='DIGEST-MD5'/>
```

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

```
<challenge xmlns='urn:ietf:params:xml:ns:xmpp-sasl'>
    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='urn:ietf:params:xml:ns:xmpp-sasl'>
    dXNlcm5hbWU9InJvYiIscmVhbG09ImNhdGFjbHlzbS5jeCIsbm9uY2U9Ik
    9BNk1HOXRFUUdtMmhoIixjbm9uY2U9Ik9BNk1IWGg2VnFUclJrIixuYz0w
    MDAwMDAwMSxxb3A9YXV0aCxkaWdlc3QtdXJpPSJ4bXBwL2NhdGFjbHlzbS
    5jeCIscmVzcG9uc2U9ZDM4OGRhZDkwZDRiYmQ3NjBhMTUyMzIxZjIxNDNh
    ZjcsY2hhcnNldD11dGYtOCxhdXRoemlkPSJyb2JAY2F0YWNseXNtLmN4L2
    15UmVzb3VyY2Ui
</response>
```

The decoded response is:

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

Step 7: Server sends another challenge to the client:

```
<challenge xmlns='urn:ietf:params:xml:ns:xmpp-sasl'>
    cnNwYXV0aD1lYTQwZjYwMzM1YzQyN2I1NTI3Yjg0ZGJhYmNkZmZmZA==
</challenge>
```

The decoded challenge is:

```
rspauth=ea40f60335c427b5527b84dbabcdfffd
```

Step 8: Client responds to the challenge:

```
<response xmlns='urn:ietf:params:xml:ns:xmpp-sasl'/>
```

Step 9: Server informs client of successful authentication:

```
<success xmlns='urn:ietf:params:xml:ns:xmpp-sasl'/>
```

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

```
<failure xmlns='urn:ietf:params:xml:ns:xmpp-sasl'>
  <mechanism-too-weak/>
</failure>
```

Step 10: Client initiates a new stream to the server:

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

Step 11: Server responds by sending a stream header to the client, with the stream already authenticated (not followed by further stream features):

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

6.1.5 Server-to-Server Example

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

Step 1: Server1 initiates stream to Server2:

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

Step 2: Server2 responds with a stream tag sent to Server1:

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

Step 3: Server2 informs Server1 of available authentication mechanisms:

```
<stream:features>
  <mechanisms xmlns='urn:ietf:params:xml:ns:xmpp-sasl'>
    <mechanism>DIGEST-MD5</mechanism>
    <mechanism>KERBEROS_V4</mechanism>
  </mechanisms>
</stream:features>
```

Step 4: Server1 selects an authentication mechanism:

```
<auth
    xmlns='urn:ietf:params:xml:ns:xmpp-sasl'
    mechanism='DIGEST-MD5'/>
```

Step 5: Server2 sends a base64-encoded challenge to Server1:

```
<challenge xmlns='urn:ietf:params:xml:ns:xmpp-sasl'>
    cmVhbG09ImNhdGFjbHlzbS5jeCIsbm9uY2U9Ik9BNk1HOXRFUUdtMmhoIi
    xxb3A9ImF1dGgiLGNoYXJzZXQ9dXRmLTgsYWxnb3JpdGhtPW1kNS1zZXNz
</challenge>
```

The decoded challenge is:

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

Step 6: Server1 responds to the challenge:

```
<response xmlns='urn:ietf:params:xml:ns:xmpp-sasl'>
    cmVhbG09ImNhdGFjbHlzbS5jeCIsbm9uY2U9Ik9BNk1HOXRFUUdtMmhoIi
    xjbm9uY2U9Ik9BNk1IWGg2VnFUclJrIixuYz0wMDAwMDAwMSxxb3A9YXV0
    aCxkaWdlc3QtdXJpPSJ4bXBwL2NhdGFjbHlzbS5jeCIscmVzcG9uc2U9ZD
    M4OGRhZDkwZDRiYmQ3NjBhMTUyMzIxZjIxNDNhZjcsY2hhcnNldD11dGYt
    OAo=
</response>
```

The decoded response is:

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

Step 7: Server2 sends another challenge to Server1:

```
<challenge xmlns='urn:ietf:params:xml:ns:xmpp-sasl'>
    cnNwYXV0aD1lYTQwZjYwMzM1YzQyN2I1NTI3Yjg0ZGJhYmNkZmZmZA==
</challenge>
```

The decoded challenge is:

```
rspauth=ea40f60335c427b5527b84dbabcdfffd
```

Step 8: Server1 responds to the challenge:

```
<response xmlns='urn:ietf:params:xml:ns:xmpp-sasl'/>
```

Step 9: Server2 informs Server1 of successful authentication:

```
<success xmlns='urn:ietf:params:xml:ns:xmpp-sasl'/>
```

Step 9 (alt): Server2 informs Server1 of failed authentication:

```
<failure xmlns='urn:ietf:params:xml:ns:xmpp-sasl'>
  <temporary-auth-failure/>
</failure>
```

Step 10: Server1 initiates a new stream to Server2:

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

Step 11: Server2 responds by sending a stream header to Server1, with the stream already authenticated (not followed by further stream features):

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

6.2 Dialback Authentication

XMPP includes a protocol-level method for verifying that a connection between two servers can be trusted 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 decidedly not intended as a mechanism for securing or encrypting the streams between servers as is done via SASL and TLS, only for helping to prevent the spoofing of a server and the sending of false data from it. In particular, dialback authentication is susceptible to DNS poisoning attacks unless DNSSec [29] is used. Furthermore, even if the DNS information is accurate, dialback authentication cannot protect from attacks where the attacker is capable of hijacking the IP address of the remote domain. Domains requiring more robust security SHOULD use TLS and SASL as defined above.

Server 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 hostname. All DNS hostname resolutions MUST first resolve the hostname using an SRV [17] 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 [5].

The method for generating and verifying the keys used in the dialback protocol MUST take into account the hostnames being used, the random ID generated for the stream, and a secret known by the authoritative server's network. Generating unique but verifiable keys is important to prevent common man-in-the-middle attacks and server spoofing.

Any error that occurs during dialback negotiation MUST be considered a stream error, resulting in termination of the stream and of the underlying TCP connection. The possible error conditions are specified in the protocol description below.

The following terminology applies:

- Originating Server -- the server that is attempting to establish a connection between two domains.

- Receiving Server -- the server that is trying to authenticate that Originating Server represents the domain which it claims to be.

- Authoritative Server -- the server that answers to the DNS

hostname asserted by Originating Server; 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.

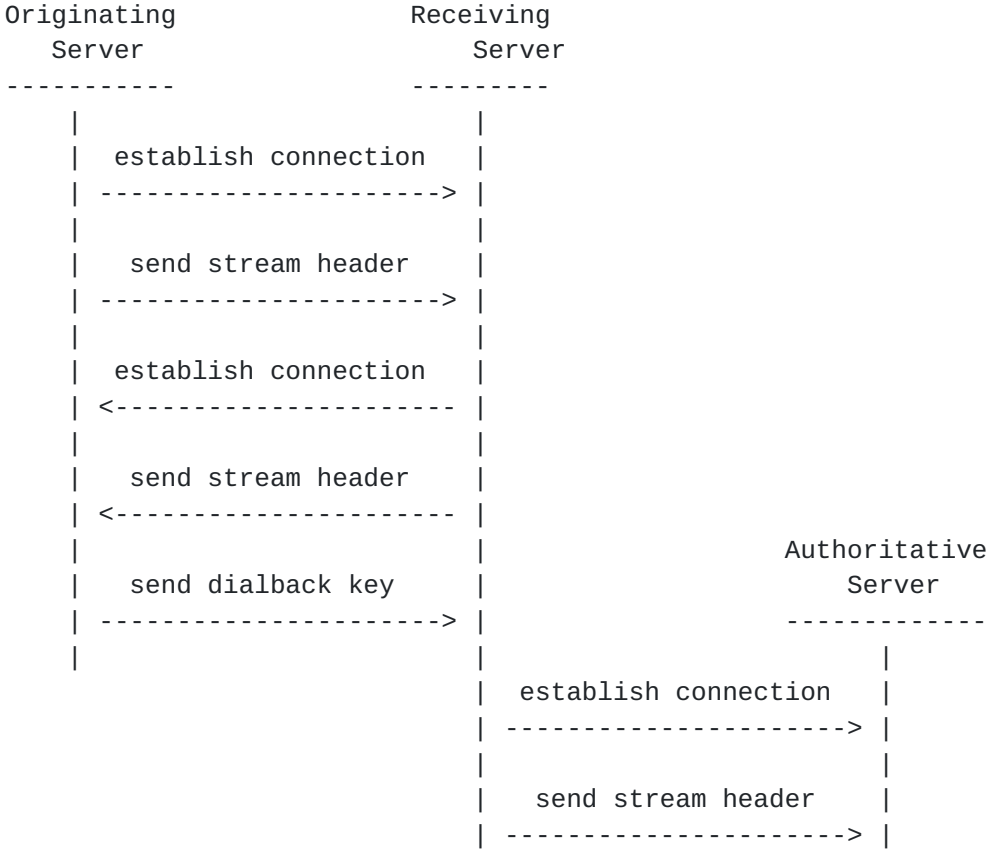
3. Receiving Server establishes a connection to Authoritative Server.

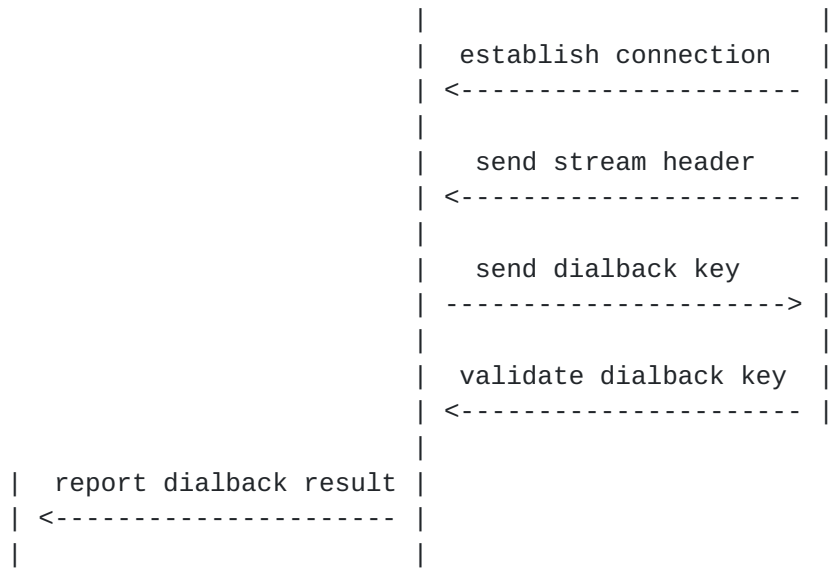
4. Receiving Server sends the same 'key' value to Authoritative Server.

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:





6.2.1 Dialback Protocol

The interaction 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:

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

Note: the 'to' and 'from' attributes are NOT REQUIRED on the root stream element. The inclusion of the xmlns:db namespace declaration with the name shown indicates to Receiving Server that Originating Server supports dialback. If the namespace name is incorrect, then Receiving Server MUST generate an <invalid-namespace/> stream error condition and terminate both the XML stream and the underlying TCP connection.

3. Receiving Server SHOULD send a stream header back to Originating Server, including a unique ID for this interaction:

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

Note: The 'to' and 'from' attributes are NOT REQUIRED on the root stream element. If the namespace name is incorrect, then Originating Server MUST generate an <invalid-namespace/> stream error condition and terminate both the XML stream and the underlying TCP connection. Note well that Receiving Server is NOT REQUIRED to reply and MAY silently terminate the XML stream and underlying TCP connection depending on security policies in place.

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. The key generated by Originating Server must be based in part on the value of the ID provided by Receiving Server in the previous step, and in part on a secret shared by Originating Server and Authoritative Server. If the value of the 'to' address does not match a hostname recognized by Receiving Server, then Receiving Server MUST generate a <host-unknown/> stream error condition and terminate both the XML stream and the underlying TCP connection. If the value of the 'from' address matches a domain with which Receiving Server already has an established connection, then Receiving Server SHOULD generate a <not-authorized/> stream error condition and terminate both the XML stream and the underlying TCP connection.

5. Receiving Server establishes a TCP connection back to the domain name asserted by Originating Server, as a result of which it connects to Authoritative Server. (Note: as an optimization, an implementation MAY reuse an existing trusted connection here rather than opening a new TCP connection.)

6. Receiving Server sends Authoritative Server a stream header:

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

Note: the 'to' and 'from' attributes are NOT REQUIRED on the root stream element. If the namespace name is incorrect, then Authoritative Server MUST generate an <invalid-namespace/> stream error condition and terminate both the XML stream and the underlying TCP connection.

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...'>
```

Note: if the namespace name is incorrect, then Receiving Server MUST generate an <invalid-namespace/> stream error condition and terminate both the XML stream and the underlying TCP connection between it and Authoritative Server. If the ID does not match that provided by Receiving Server in Step 3, then Receiving Server MUST generate an <invalid-id/> stream error condition and terminate both the XML stream and the underlying TCP connection between it and Authoritative Server. If either of the foregoing stream errors occurs between Receiving Server and Authoritative Server, then Receiving Server MUST generate a <remote-connection-failed/> stream error condition and terminate both the XML stream and the underlying TCP connection between it and Originating Server.

8. Receiving Server sends Authoritative Server a stanza requesting that Authoritative Server 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 3, and the key that Originating Server sent to Receiving Server in Step 4. Based on this information and shared secret information within the Authoritative Server's network, the key

is verified. Any verifiable method MAY be used to generate the key. If the value of the 'to' address does not match a hostname recognized by Authoritative Server, then Authoritative Server MUST generate a <host-unknown/> stream error condition and terminate both the XML stream and the underlying TCP connection. If the value of the 'from' address does not match the hostname represented by Receiving Server when opening the TCP connection (or any validated domain), then Authoritative Server MUST generate a <nonmatching-hosts/> stream error condition and terminate both the XML stream and the underlying TCP connection.

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...'/>
```

Note: if the ID does not match that provided by Receiving Server in Step 3, then Receiving Server MUST generate an <invalid-id/> stream error condition and terminate both the XML stream and the underlying TCP connection. If the value of the 'to' address does not match a hostname recognized by Receiving Server, then Receiving Server MUST generate a <host-unknown/> stream error condition and terminate both the XML stream and the underlying TCP connection. If the value of the 'from' address does not match the hostname represented by Originating Server when opening the TCP connection (or any validated domain), then Receiving Server MUST generate a <nonmatching-hosts/> stream error condition and terminate both the XML stream and the underlying TCP connection.

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. If the connection is invalid, then Receiving Server MUST terminate both the XML stream and the underlying TCP connection. If 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 silently dropped.

Even if dialback negotiation is successful, a server MUST verify that all XML stanzas received from the other server include a 'from' attribute and a 'to' attribute; if a stanza does not meet this restriction, the server that receives the stanza MUST generate an <invalid-xml/> stream error condition and terminate both the XML stream and the underlying TCP connection. Furthermore, a server MUST verify that the 'from' attribute of stanzas received from the other server includes the validated domain (or any validated domain); if a stanza does not meet this restriction, the server that receives the stanza MUST generate a <nonmatching-hosts/> stream error condition and terminate both the XML stream and the underlying TCP connection. Both of these checks help to prevent spoofing related to particular stanzas.

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. Three XML stanza types are defined for the 'jabber:client' and 'jabber:server' namespaces: <message/>, <presence/>, and <iq/>.

In essence, the <message/> stanza type can be seen as a "push" mechanism whereby one entity pushes information to another entity, similar to the communications that occur in a system such as email. The <presence/> element can be seen as a basic broadcast or "publish-subscribe" mechanism, whereby multiple entities receive information (in this case, presence information) about an entity to which they have subscribed. The <iq/> element can be seen as a "request-response" mechanism similar to HTTP, whereby two entities can engage in a structured conversation using 'get' or 'set' requests and 'result' or 'error' responses.

The syntax for these stanza types is defined below.

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; if a server receives a stanza that does not meet this restriction, it MUST generate an <invalid-xml/> stream error condition and terminate both the XML stream and the underlying TCP connection.

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 and MAY return an error to the sender. 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; if a server receives a stanza that does not meet this restriction, it MUST generate an <invalid-xml/> stream error condition. Furthermore, the domain identifier portion of the JID contained in the 'from' attribute MUST match the hostname of the sending server (or any validated domain) as communicated in the SASL negotiation or dialback negotiation; if a server receives a stanza that does not meet this restriction, it MUST generate a <nonmatching-hosts/> stream error condition. Both of these conditions MUST result in closing of the stream and termination of the underlying TCP connection.

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 [16].

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; if included, it 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:

- chat
- error
- groupchat
- headline

For information about the meaning of these message types, refer to XMPP IM [22].

7.3.2 Children

As described under extended namespaces (Section 7.6), a message stanza MAY contain any properly-namespaced child element as long as the namespace name is not "jabber:client", "jabber:server", or "http://etherx.jabber.org/streams".

In accordance with the default namespace declaration, by default a message stanza is in the 'jabber:client' or 'jabber:server' namespace, which defines certain allowable children of message stanzas. If the message stanza is of type "error", it MUST include an <error/> child; for details, see Section 7.7. If the message stanza 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 without an explicit namespace declaration:

7.3.2.1 Body

The <body/> element contains the textual contents of the message; normally included but NOT REQUIRED. The <body/> element SHOULD 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. The <body> element MUST NOT contain mixed content.

7.3.2.2 Subject

The <subject/> element specifies the topic of the message. The <subject/> element SHOULD 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. The <subject> element MUST NOT contain mixed content.

7.3.2.3 Thread

The <thread/> element contains a random string that is generated by the sender and that SHOULD 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. 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 be made against it. The <thread> element MUST NOT contain mixed content.

The method for generating thread IDs SHOULD be as follows:

1. concatenate the sender's full JID (user@domain/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 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

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 text) and to communicate that status to other entities. Presence stanzas 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 possess 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 a lack of availability, 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 [22].

7.4.2 Children

As described under extended namespaces (Section 7.6), a presence stanza MAY contain any properly-namespaced child element as long as

the namespace name is not "jabber:client", "jabber:server", or "http://etherx.jabber.org/streams".

In accordance with the default namespace declaration, by default a presence stanza is in the 'jabber:client' or 'jabber:server' namespace, which defines certain allowable children of presence stanzas. If the presence stanza is of type "error", it MUST include an <error/> child; for details, see Section 7.7. If the 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"):

7.4.2.1 Show

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

- away
- chat
- xa
- dnd

For information about the meaning of these values, refer to XMPP IM [22].

7.4.2.2 Status

The optional <status/> element contains a natural-language description of availability status. It is 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 SHOULD 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.

7.4.2.3 Priority

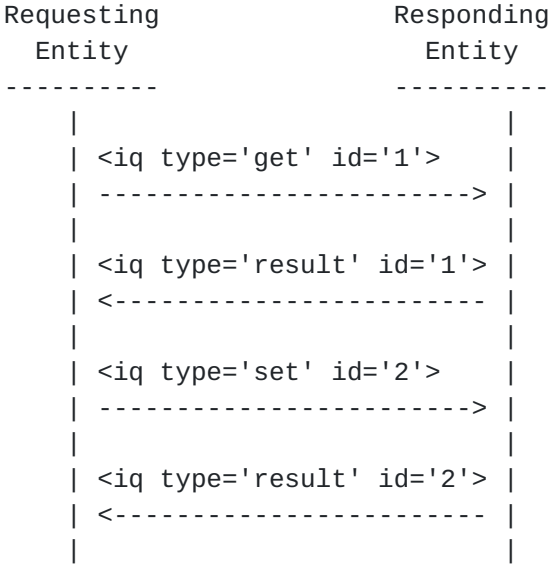
The optional <priority/> element specifies the priority level of the connected resource. The value may be any integer between -128 to 127. Only one <priority/> element MAY be included in a presence stanza, and it MUST NOT possess any attributes. For information regarding the use of priority values in stanza routing within IM applications, see XMPP IM [22].

7.5 IQ Stanzas

7.5.1 Overview

Info/Query, or IQ, is a request-response mechanism, similar in some ways to HTTP [30]. 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 MUST 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 SHOULD 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

As described under extended namespaces (Section 7.6), an IQ stanza MAY contain any properly-namespaced child element as long as the namespace name is not "jabber:client", "jabber"server", or "http://etherx.jabber.org/streams". However, an IQ stanza contains no children in the 'jabber:client' or 'jabber:server' namespace since it is a vessel for XML in another namespace.

An IQ stanza of type "get" or "set" MUST include one and only one child element. An the IQ stanza of type "error" SHOULD include the child element contained in the associated "set" or "get" and MUST include an <error/> child; for details, see Section 7.7.

7.6 Extended Namespaces

While the core data elements in the "jabber:client" or "jabber:server" namespace (along with their attributes and child elements) 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 have any name and MUST possess an 'xmlns' namespace

declaration (other than "jabber:client", "jabber:server", or "http://etherx.jabber.org/streams") 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, the entity's expected behavior depends on whether the entity is (1) the recipient or (2) an entity that is routing the stanza to the recipient. In particular:

Recipient: If a recipient receives a stanza that contains a child element it does not understand, it SHOULD ignore that specific XML data, i.e., it SHOULD not process it or present it to a user or associated application (if any). In particular:

* 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/

* 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 condition of <feature-not-implemented/>.

Router: If a routing entity (usually a server) handles a stanza that contains a child element it does not understand, it SHOULD ignore the associated XML data by passing it on untouched to the recipient.

7.7 Stanza Errors

As defined below, stanza-related errors are handled in a manner similar to stream errors (Section 4.5).

7.7.1 Rules

The following rules apply to stanza-related errors:

o A stanza of type "error" MUST contain an <error/> child element.

o The receiving or processing entity that returns an error to the sending entity SHOULD include the original XML sent along with the

<error/> element and its children so that the sender can inspect and if necessary correct the XML before re-sending.

- 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.

- An <error/> child MUST NOT be included if the stanza type is something other than "error".

7.7.2 Syntax

The syntax for stanza-related errors is as follows:

```
<stanza-name to='sender' type='error'>
  [include sender XML here]
  <error class='error-class'>
    <condition xmlns='urn:ietf:params:xml:ns:xmpp-stanzas'>
      <descriptive-element-name/>
    </condition>
  </error>
</stanza-name>
```

The stanza-name is one of message, presence, or iq.

The value of the 'class' attribute MUST be one of the following:

- access -- the condition relates to access rights, permissions, or authorization

- address -- the condition relates to the JID or domain to which the stanza was addressed

- app -- the condition is particular to an application and is specified in a namespace other than 'urn:ietf:params:xml:ns:xmpp-stanzas'

- format -- the condition relates to XML format or structure

- recipient -- the condition relates to the state or capabilities of the recipient (which may be the server)

- server -- the condition relates to the internal state of the server

The <condition/> element MUST contain a child element that specifies a particular stanza-related error condition, as defined in the next

section. (Note: the XML namespace name 'urn:ietf:params:xml:ns:xmpp-stanzas' that scopes the <condition/> element adheres to the format defined in The IETF XML Registry [24].)

7.7.3 Conditions

The following stanza-related error conditions are defined:

- <bad-request/> -- the sender has sent XML that is malformed or cannot be processed (e.g., a client-generated stanza includes a 'from' address, or an IQ stanza includes an unrecognized value of the 'type' attribute); the associated class is "format".

- <conflict/> -- access cannot be granted because an existing resource or session exists with the same name or address; the associated class is "access".

- <feature-not-implemented/> -- the feature requested is not implemented by the recipient or server and therefore cannot be processed; the associated class is "recipient".

- <forbidden/> -- the stanza is understood but the action is forbidden; the associated class is "access".

- <internal-server-error/> -- the server could not process the stanza because of a misconfiguration or an otherwise-undefined internal server error; the associated class is "server".

- <jid-malformed/> -- the value of the 'to' attribute in the sender's stanza does not adhere to the syntax defined in Addressing (Section 3); the associated class is "address".

- <jid-not-found/> -- the value of the 'to' attribute in the sender's stanza does not correspond to a Jabber ID on the user's server or a remote server; the associated class is "address".

- <not-allowed/> -- the action is not permitted when attempted by the sender; the associated class is "access".

- <recipient-unavailable/> -- the specific recipient requested is currently unavailable; the associated class is "recipient".

- <registration-required/> -- the user is not authorized to access the requested service because registration is required; the associated class is "access".

- <remote-server-not-found/> -- a remote server or service specified as part or all of the JID of the intended recipient does not

exist; the associated class is "address".

- <remote-server-timeout/> -- a remote server or service specified as part or all of the JID of the intended recipient could not be contacted within a reasonable amount of time; the associated class is "server".

- <service-unavailable/> -- the service requested is currently unavailable on the server; the associated class is "server".

7.7.4 Extensibility

If desired, an XMPP application MAY provide custom error information; the error class MUST be "app" and the data MUST be contained in a properly-namespaced child of the <condition/> element (i.e., the namespace name MUST NOT be one of namespace names defined herein).

8. XML Usage within XMPP

8.1 Restrictions

XMPP is a simplified and specialized protocol for streaming XML elements in order to exchange messages and presence information in close to real time. Because XMPP does not require the parsing of arbitrary and complete XML documents, there is no requirement that XMPP must support the full XML specification [1]. In particular, the following restrictions apply:

With regard to XML generation, an XMPP implementation MUST NOT inject into an XML stream any of the following:

- comments (as defined in Section 2.5 of the XML specification [1])
- processing instructions (Section 2.6)
- internal or external DTD subsets (Section 2.8)
- internal or external entity references (Section 4.2) with the exception of predefined entities (Section 4.6)

With regard to XML processing, if an XMPP implementation receives such restricted XML data, it MUST ignore the data.

8.2 Namespaces

XML Namespaces [12] 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.

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

8.3 Validation

Except as noted with regard to 'to' and 'from' addresses for stanzas within the 'jabber:server' namespace, a server is not responsible for validating the XML elements forwarded to a client or another server; 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 schemas are included herein for descriptive purposes only.

8.4 Character Encodings

Software implementing XML streams MUST support the UTF-8 (RFC 2279 [18]) and UTF-16 (RFC 2781 [19]) transformations of Universal Character Set (ISO/IEC 10646-1 [20]) 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.

8.5 Inclusion of Text Declaration

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

9. IANA Considerations

9.1 XML Namespace Name for TLS Data

A URN sub-namespace for TLS-related data in the Extensible Messaging and Presence Protocol (XMPP) is defined as follows.

URI: urn:ietf:params:xml:ns:xmpp-tls

Specification: [RFCXXXX]

Description: This is the XML namespace name for TLS-related data in the Extensible Messaging and Presence Protocol (XMPP) as defined by [RFCXXXX].

Registrant Contact: IETF, XMPP Working Group, <xmppwg@jabber.org>

9.2 XML Namespace Name for SASL Data

A URN sub-namespace for SASL-related data in the Extensible Messaging and Presence Protocol (XMPP) is defined as follows.

URI: urn:ietf:params:xml:ns:xmpp-sasl

Specification: [RFCXXXX]

Description: This is the XML namespace name for SASL-related data in the Extensible Messaging and Presence Protocol (XMPP) as defined by [RFCXXXX].

Registrant Contact: IETF, XMPP Working Group, <xmppwg@jabber.org>

9.3 XML Namespace Name for Stream Errors

A URN sub-namespace for stream-related error data in the Extensible Messaging and Presence Protocol (XMPP) is defined as follows.

URI: urn:ietf:params:xml:ns:xmpp-streams

Specification: [RFCXXXX]

Description: This is the XML namespace name for stream-related error data in the Extensible Messaging and Presence Protocol (XMPP) as defined by [RFCXXXX].

Registrant Contact: IETF, XMPP Working Group, <xmppwg@jabber.org>

9.4 XML Namespace Name for Stanza Errors

A URN sub-namespace for stanza-related error data in the Extensible Messaging and Presence Protocol (XMPP) is defined as follows.

URI: urn:ietf:params:xml:ns:xmpp-stanzas

Specification: [RFCXXXX]

Description: This is the XML namespace name for stanza-related error data in the Extensible Messaging and Presence Protocol (XMPP) as defined by [RFCXXXX].

Registrant Contact: IETF, XMPP Working Group, <xmppwg@jabber.org>

9.5 Existing Registrations

The IANA registers "xmpp" as a GSSAPI [21] service name, as specified in Section 6.1.3.

Additionally, the IANA registers "jabber-client" and "jabber-server" as keywords for 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, a server MUST NOT modify or delete it.

11. Security Considerations

11.1 High Security

For the purposes of XMPP communications (client-to-server and server-to-server), the term "high security" refers to the use of security technologies that provide both mutual authentication and integrity-checking; in particular, when using certificate-based authentication to provide high security, a chain-of-trust SHOULD be established out-of-band, although a shared certificate authority signing certificates could allow a previously unknown certificate to establish trust in-band.

Self-signed certificates MAY be used but pose a problem for administrators the first time such a certificate is seen. A self-signed certificate, if accepted, MUST be stored by an entity in order to verify in future communications. A server that changes its self-signed cert to another self-signed cert (or to a certificate signed by an unrecognized authority) therefore creates administration problems for all entities with which it has communicated before and will again. In particular, those entities have no reason to believe that the new self-signed cert was not generated by an attacker to impersonate the previously-trusted server.

Implementations MUST support high security. Service provisioning SHOULD use high security, subject to local security policies.

11.2 Client-to-Server Communications

The TLS protocol for encrypting XML streams (defined under Section 5) provides a reliable mechanism for helping to ensure the confidentiality and data integrity of data exchanged between two entities.

The SASL protocol for authenticating XML streams (defined under Section 6.1) 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 [31].

11.3 Server-to-Server Communications

A compliant implementation MUST support both TLS and SASL for inter-domain communications. For historical reasons, a compliant implementation SHOULD also support the lower-security Dialback Protocol (Section 6.2), which provides a mechanism for helping to prevent the spoofing of domains.

Because service provisioning is a matter of policy, it is OPTIONAL for any given domain to communicate with other domains, and server-to-server communications MAY be disabled by the administrator of any given deployment. If a particular domain enables inter-domain communications, it SHOULD enable high security. In the absence of high security, a domain MAY use server dialback for inter-domain communications.

Administrators may want to require use of SASL for server-to-server communications in order to ensure authentication and confidentiality (e.g., on an organization's private network). Compliant implementations SHOULD support SASL for this purpose.

11.4 Firewalls

Communications using XMPP normally occur over TCP sockets on port 5222 (client-to-server) or port 5269 (server-to-server), as registered with the IANA [5]. Use of these well-known ports allows administrators to easily enable or disable XMPP activity through existing and commonly-deployed firewalls.

11.5 Mandatory to Implement Technologies

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)

Normative 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] 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>.

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

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

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

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

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

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

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

[10] Saint-Andre, P. and J. Hildebrand, "Nodeprep: A Stringprep Profile for Node Identifiers in XMPP (draft-ietf-xmpp-nodeprep-02, work in progress)", April 2003.

[11] Saint-Andre, P. and J. Hildebrand, "Resourceprep: A Stringprep Profile for Resource Identifiers in XMPP (draft-ietf-xmpp-resourceprep-02, work in progress)", April 2003.

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

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

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

[15] Leach, P. and C. Newman, "Using Digest Authentication as a SASL Mechanism", RFC 2831, May 2000.

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

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

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

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

[20] 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.

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

Informative References

[22] Saint-Andre, P. and J. Miller, "XMPP Instant Messaging (draft-ietf-xmpp-im-09, work in progress)", April 2003.

[23] 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>.

[24] Mealling, M., "The IANA XML Registry", draft-mealling-iana-xmlns-registry-04 (work in progress), June 2002.

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

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

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

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

[29] Eastlake, D., "Domain Name System Security Extensions", RFC 2535, March 1999.

[30] Fielding, R., Gettys, J., Mogul, J., Nielsen, H., Masinter, L., Leach, P. and T. Berners-Lee, "Hypertext Transfer Protocol -- HTTP/1.1", RFC 2616, June 1999.

[31] Saint-Andre, P. and J. Hildebrand, "End-to-End Object Encryption in XMPP (draft-ietf-xmpp-e2e-02, work in progress)", April 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. XML Schemas

The following XML schemas are descriptive, not normative.

A.1 Streams namespace

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

<xs:schema
    xmlns:xs='http://www.w3.org/2001/XMLSchema'
    targetNamespace='http://etherx.jabber.org/streams'
    xmlns='http://etherx.jabber.org/streams'
    elementFormDefault='qualified'>

  <xs:element name='stream'>
    <xs:complexType>
      <xs:element ref='features' minOccurs='0' maxOccurs='1'/>
      <xs:element ref='error' minOccurs='0' maxOccurs='1'/>
      <xs:choice>
        <xs:any
             namespace='jabber:client'
             maxOccurs='1'/>
        <xs:any
             namespace='jabber:server'
             maxOccurs='1'/>
      </xs:choice>
      <xs:attribute name='to' type='xs:string' use='optional'/>
      <xs:attribute name='from' type='xs:string' use='optional'/>
      <xs:attribute name='id' type='xs:ID' use='optional'/>
      <xs:attribute name='version' type='xs:decimal' use='optional'/>
    </xs:complexType>
  </xs:element>

  <xs:element name='features'>
    <xs:complexType>
      <xs:any
          namespace='##other'
          minOccurs='0'
          maxOccurs='unbounded'/>
    </xs:complexType>
  </xs:element>

  <xs:element name='error'>
    <xs:complexType>
      <xs:attribute name='class' use='required'/>
        <xs:simpleType>
          <xs:restriction base='xs:NCName'>
            <xs:enumeration value='address'/>
```

```
              <xs:enumeration value='app'/>
              <xs:enumeration value='format'/>
              <xs:enumeration value='redirect'/>
              <xs:enumeration value='server'/>
            </xs:restriction>
          </xs:simpleType>
        </xs:attribute>
      </xs:complexType>
    </xs:element>

  </xs:schema>
```

A.2 TLS namespace

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

  <xs:schema
      xmlns:xs='http://www.w3.org/2001/XMLSchema'
      targetNamespace='urn:ietf:params:xml:ns:xmpp-tls'
      xmlns='urn:ietf:params:xml:ns:xmpp-tls'
      elementFormDefault='qualified'>

    <xs:element name='starttls'>
      <xs:complexType>
        <xs:element ref='required' maxOccurs='1'>
      </xs:complexType>
    </xs:element>
    <xs:element name='proceed'/>
    <xs:element name='failure'/>
    <xs:element name='required'/>

  </xs:schema>
```

A.3 SASL namespace

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

  <xs:schema
      xmlns:xs='http://www.w3.org/2001/XMLSchema'
      targetNamespace='urn:ietf:params:xml:ns:xmpp-sasl'
      xmlns='urn:ietf:params:xml:ns:xmpp-sasl'
      elementFormDefault='qualified'>

    <xs:element name='mechanisms'>
      <xs:complexType>
        <xs:element ref='mechanism' maxOccurs='unbounded'>
```

```
      </xs:complexType>
    </xs:element>

    <xs:element name='mechanism' type='xs:string'/>

    <xs:element name='auth'>
      <xs:complexType>
        <xs:attribute name='mechanism'
                      type='xs:string' use='optional'/>
      </xs:complexType>
    </xs:element>

    <xs:element name='challenge' type='xs:string'/>
    <xs:element name='response' type='xs:string'/>
    <xs:element name='abort'/>
    <xs:element name='success'/>
    <xs:element name='failure'>
      <xs:complexType>
        <xs:choice>
          <xs:element ref='invalid-realm' maxOccurs='1'>
          <xs:element ref='mechanism-too-weak' maxOccurs='1'>
          <xs:element ref='not-authorized' maxOccurs='1'>
          <xs:element ref='password-transition-required' maxOccurs='1'>
          <xs:element ref='temporary-auth-failure' maxOccurs='1'>
        </xs:choice>
      </xs:complexType>
    </xs:element>

    <xs:element name='invalid-realm'/>
    <xs:element name='mechanism-too-weak'/>
    <xs:element name='not-authorized'/>
    <xs:element name='password-transition-required'/>
    <xs:element name='temporary-auth-failure'/>

  </xs:schema>
```

A.4 Dialback namespace

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

  <xs:schema
      xmlns:xs='http://www.w3.org/2001/XMLSchema'
      targetNamespace='jabber:server:dialback'
      xmlns='jabber:server:dialback'
      elementFormDefault='qualified'>

    <xs:element name='result'>
```

```
      <xs:complexType>
        <xs:simpleContent>
          <xs:extension base='xs:string'>
            <xs:attribute name='from' type='xs:string' use='required'/>
            <xs:attribute name='to' type='xs:string' use='required'/>
            <xs:attribute name='type' use='optional'>
              <xs:simpleType>
                <xs:restriction base='xs:NCName'>
                  <xs:enumeration value='invalid'/>
                  <xs:enumeration value='valid'/>
                </xs:restriction>
              </xs:simpleType>
            </xs:attribute>
          </xs:extension>
        </xs:simpleContent>
      </xs:complexType>
    </xs:element>

    <xs:element name='verify'>
      <xs:complexType>
        <xs:simpleContent>
          <xs:extension base='xs:string'>
            <xs:attribute name='from' type='xs:string' use='required'/>
            <xs:attribute name='to' type='xs:string' use='required'/>
            <xs:attribute name='id' type='xs:string' use='required'/>
            <xs:attribute name='type' use='optional'>
              <xs:simpleType>
                <xs:restriction base='xs:NCName'>
                  <xs:enumeration value='invalid'/>
                  <xs:enumeration value='valid'/>
                </xs:restriction>
              </xs:simpleType>
            </xs:attribute>
          </xs:extension>
        </xs:simpleContent>
      </xs:complexType>
    </xs:element>

  </xs:schema>
```

A.5 Client namespace

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

  <xs:schema
      xmlns:xs='http://www.w3.org/2001/XMLSchema'
      targetNamespace='jabber:client'
```

```
       xmlns='jabber:client'
       elementFormDefault='qualified'>

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

     <xs:element name='body' type='xs:string'>
       <xs:complexType>
         <xs:attribute name='xml:lang'
                       type='xmlLangType' use='optional'/>
       </xs:complexType>
     </xs:element>

     <xs:element name='subject' type='xs:string'>
       <xs:complexType>
         <xs:attribute name='xml:lang'
                       type='xmlLangType' use='optional'/>
       </xs:complexType>
     </xs:element>
```

```
    <xs:element name='thread' type='xs:string'/>

    <xs:element name='presence'>
      <xs:complexType>
        <xs:choice maxOccurs='unbounded'>
          <xs:element ref='show' minOccurs='0' maxOccurs='1'/>
          <xs:element ref='status'
                      minOccurs='0' maxOccurs='unbounded'/>
          <xs:element ref='priority' minOccurs='0' maxOccurs='1'/>
          <xs:element ref='error' minOccurs='0' maxOccurs='1'/>
          <xs:any
              namespace='##other'
              minOccurs='0'
              maxOccurs='unbounded'/>
        </xs:choice>
        <xs:attribute name='to' type='xs:string' use='optional'/>
        <xs:attribute name='from' type='xs:string' use='optional'/>
        <xs:attribute name='id' type='xs:ID' use='optional'/>
        <xs:attribute name='xml:lang'
                      type='xmlLangType' use='optional'/>
        <xs:attribute name='type' use='optional'>
          <xs:simpleType>
            <xs:restriction base='xs:NCName'>
              <xs:enumeration value='subscribe'/>
              <xs:enumeration value='subscribed'/>
              <xs:enumeration value='unsubscribe'/>
              <xs:enumeration value='unsubscribed'/>
              <xs:enumeration value='unavailable'/>
              <xs:enumeration value='error'/>
            </xs:restriction>
          </xs:simpleType>
        </xs:attribute>
      </xs:complexType>
    </xs:element>

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

    <xs:element name='status' type='xs:string'>
      <xs:complexType>
```

```
      <xs:attribute name='xml:lang'
                    type='xmlLangType' use='optional'/>
    </xs:complexType>
  </xs:element>

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

  <xs:element name='iq'>
    <xs:complexType>
      <xs:sequence>
        <xs:any
            namespace='##other'
            minOccurs='0'
            maxOccurs='1'/>
        <xs:element ref='error' minOccurs='0' maxOccurs='1'/>
      </xs:sequence>
      <xs:attribute name='to' type='xs:string' use='optional'/>
      <xs:attribute name='from' type='xs:string' use='optional'/>
      <xs:attribute name='id' type='xs:ID' use='optional'/>
      <xs:attribute name='type' use='required'>
        <xs:simpleType>
          <xs:restriction base='xs:NCName'>
            <xs:enumeration value='get'/>
            <xs:enumeration value='set'/>
            <xs:enumeration value='result'/>
            <xs:enumeration value='error'/>
          </xs:restriction>
        </xs:simpleType>
      </xs:attribute>
    </xs:complexType>
  </xs:element>

  <xs:element name='error'>
    <xs:complexType>
      <xs:attribute name='class' use='required'/>
        <xs:simpleType>
          <xs:restriction base='xs:NCName'>
            <xs:enumeration value='access'/>
            <xs:enumeration value='address'/>
            <xs:enumeration value='app'/>
            <xs:enumeration value='format'/>
            <xs:enumeration value='recipient'/>
            <xs:enumeration value='server'/>
          </xs:restriction>
        </xs:simpleType>
      </xs:attribute>
    </xs:complexType>
  </xs:element>
```

```
    <xs:simpleType name="xmlLangType">
      <xs:restriction base="xs:NMTOKEN">
        <xs:pattern value="[a-z]{2}-[A-Z]{2}"/>
      </xs:restriction>
    </xs:simpleType>

  </xs:schema>
```

A.6 Server namespace

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

  <xs:schema
      xmlns:xs='http://www.w3.org/2001/XMLSchema'
      targetNamespace='http://www.jabber.org/protocol'
      xmlns='http://www.jabber.org/protocol'
      elementFormDefault='qualified'>

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

```
       </xs:complexType>
    </xs:element>

    <xs:element name='body' type='xs:string'>
      <xs:complexType>
        <xs:attribute name='xml:lang'
                      type='xmlLangType' use='optional'/>
      </xs:complexType>
    </xs:element>

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

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

    <xs:element name='presence'>
      <xs:complexType>
        <xs:choice maxOccurs='unbounded'>
          <xs:element ref='show' minOccurs='0' maxOccurs='1'/>
          <xs:element ref='status'
                      minOccurs='0' maxOccurs='unbounded'/>
          <xs:element ref='priority' minOccurs='0' maxOccurs='1'/>
          <xs:element ref='error' minOccurs='0' maxOccurs='1'/>
          <xs:any
              namespace='##other'
              minOccurs='0'
              maxOccurs='unbounded'/>
        </xs:choice>
        <xs:attribute name='to' type='xs:string' use='required'/>
        <xs:attribute name='from' type='xs:string' use='required'/>
        <xs:attribute name='id' type='xs:ID' use='optional'/>
        <xs:attribute name='xml:lang'
                      type='xmlLangType' use='optional'/>
        <xs:attribute name='type' use='optional'>
          <xs:simpleType>
            <xs:restriction base='xs:NCName'>
              <xs:enumeration value='subscribe'/>
              <xs:enumeration value='subscribed'/>
              <xs:enumeration value='unsubscribe'/>
              <xs:enumeration value='unsubscribed'/>
              <xs:enumeration value='unavailable'/>
              <xs:enumeration value='error'/>
            </xs:restriction>
          </xs:simpleType>
```

```
        </xs:attribute>
      </xs:complexType>
    </xs:element>

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

    <xs:element name='status' type='xs:string'>
      <xs:complexType>
        <xs:attribute name='xml:lang'
                      type='xmlLangType' use='optional'/>
      </xs:complexType>
    </xs:element>

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

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

```
    <xs:element name='error'>
      <xs:complexType>
        <xs:attribute name='class' use='required'/>
          <xs:simpleType>
            <xs:restriction base='xs:NCName'>
              <xs:enumeration value='access'/>
              <xs:enumeration value='address'/>
              <xs:enumeration value='app'/>
              <xs:enumeration value='format'/>
              <xs:enumeration value='recipient'/>
              <xs:enumeration value='server'/>
            </xs:restriction>
          </xs:simpleType>
        </xs:attribute>
      </xs:complexType>
    </xs:element>

    <xs:simpleType name="xmlLangType">
      <xs:restriction base="xs:NMTOKEN">
        <xs:pattern value="[a-z]{2}-[A-Z]{2}"/>
      </xs:restriction>
    </xs:simpleType>

  </xs:schema>
```

A.7 Stream error namespace

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

  <xs:schema
      xmlns:xs='http://www.w3.org/2001/XMLSchema'
      targetNamespace='urn:ietf:params:xml:ns:xmpp-streams'
      xmlns='urn:ietf:params:xml:ns:xmpp-streams'
      elementFormDefault='qualified'>

    <xs:element name='condition'>
      <xs:complexType>
        <xs:any
            namespace='##other'
            minOccurs='0'
            maxOccurs='1'/>
        <xs:choice maxOccurs='1'>
          <xs:element ref='host-gone'/>
          <xs:element ref='host-unknown'/>
          <xs:element ref='internal-server-error'/>
          <xs:element ref='invalid-id'/>
          <xs:element ref='invalid-namespace'/>
```

```
          <xs:element ref='nonmatching-hosts'/>
          <xs:element ref='not-authorized'/>
          <xs:element ref='remote-connection-failed'/>
          <xs:element ref='resource-constraint'/>
          <xs:element ref='see-other-host'/>
          <xs:element ref='system-shutdown'/>
          <xs:element ref='unsupported-stanza-type'/>
          <xs:element ref='unsupported-version'/>
          <xs:element ref='xml-not-well-formed'/>
        </xs:choice>
      </xs:complexType>

    <xs:element name='host-gone'/>
    <xs:element name='host-unknown'/>
    <xs:element name='internal-server-error'/>
    <xs:element name='invalid-id'/>
    <xs:element name='invalid-namespace'/>
    <xs:element name='nonmatching-hosts'/>
    <xs:element name='not-authorized'/>
    <xs:element name='remote-connection-failed'/>
    <xs:element name='resource-constraint'/>
    <xs:element name='see-other-host'/>
    <xs:element name='system-shutdown'/>
    <xs:element name='unsupported-stanza-type'/>
    <xs:element name='unsupported-version'/>
    <xs:element name='xml-not-well-formed'/>

  </xs:schema>
```

A.8 Stanza error namespace

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

  <xs:schema
      xmlns:xs='http://www.w3.org/2001/XMLSchema'
      targetNamespace='urn:ietf:params:xml:ns:xmpp-stanzas'
      xmlns='urn:ietf:params:xml:ns:xmpp-stanzas'
      elementFormDefault='qualified'>

    <xs:element name='condition'>
      <xs:complexType>
        <xs:any
            namespace='##other'
            minOccurs='0'
            maxOccurs='1'/>
        <xs:choice maxOccurs='1'>
          <xs:element ref='bad-request'/>
```

```
          <xs:element ref='feature-not-implemented'/>
          <xs:element ref='forbidden'/>
          <xs:element ref='internal-server-error'/>
          <xs:element ref='jid-malformed'/>
          <xs:element ref='jid-not-found'/>
          <xs:element ref='not-allowed'/>
          <xs:element ref='recipient-unavailable'/>
          <xs:element ref='registration-required'/>
          <xs:element ref='remote-server-not-found'/>
          <xs:element ref='remove-server-timeout'/>
          <xs:element ref='service-unavailable'/>
        </xs:choice>
      </xs:complexType>

    <xs:element name='bad-request'/>
    <xs:element name='feature-not-implemented'/>
    <xs:element name='forbidden'/>
    <xs:element name='internal-server-error'/>
    <xs:element name='jid-malformed'/>
    <xs:element name='jid-not-found'/>
    <xs:element name='not-allowed'/>
    <xs:element name='recipient-unavailable'/>
    <xs:element name='registration-required'/>
    <xs:element name='remote-server-not-found'/>
    <xs:element name='remote-server-timeout'/>
    <xs:element name='service-unavailable'/>

  </xs:schema>
```

Appendix B. Revision History

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

B.1 Changes from draft-ietf-xmpp-core-09

- Fixed several dialback error conditions.
- Changed <stream-condition/> and <stanza-condition/> elements to <condition/>.
- Added or modified several stream and stanza error conditions.
- Specified only one child allowed for IQ, or two if type="error".

B.2 Changes from draft-ietf-xmpp-core-08

- Incorporated list discussion regarding addressing, SASL, TLS, TCP, dialback, namespaces, extensibility, and the meaning of 'ignore' for routers and recipients.
- Specified dialback error conditions.
- Made small editorial changes to address RFC Editor requirements.

B.3 Changes from draft-ietf-xmpp-core-07

- Made several small editorial changes.

B.4 Changes from draft-ietf-xmpp-core-06

- Added text regarding certificate validation in TLS negotiation per list discussion.
- Clarified nature of XML restrictions per discussion with W3C, and moved XML Restrictions subsection under "XML Usage within XMPP".
- Further clarified that XML streams are unidirectional.
- Changed stream error and stanza error namespace names to conform to the format defined in The IETF XML Registry [24].
- Removed note to RFC Editor regarding provisional namespace names.

B.5 Changes from draft-ietf-xmpp-core-05

- Added <invalid-namespace/> as a stream error condition.
- Adjusted security considerations per discussion at IETF 56 and on list.

B.6 Changes from draft-ietf-xmpp-core-04

- Added server-to-server examples for TLS and SASL.
- Changed error syntax, rules, and examples based on list discussion.
- Added schemas for the TLS, stream error, and stanza error namespaces.
- Added note to RFC Editor regarding provisional namespace names.
- Made numerous small editorial changes and clarified text throughout.

B.7 Changes from draft-ietf-xmpp-core-03

- Clarified rules and procedures for TLS and SASL.
- Amplified stream error code syntax per list discussion.
- Made numerous small editorial changes.

B.8 Changes from draft-ietf-xmpp-core-02

- Added dialback schema.
- Removed all DTDs since schemas provide more complete definitions.
- Added stream error codes.
- Clarified error code "philosophy".

B.9 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.

B.10 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.

B.11 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.