XMPP Core
draft-ietf-xmpp-core-06

Status of this Memo

Copyright Notice

Abstract

   This document describes the core features of the Extensible Messaging
   and Presence Protocol (XMPP), a protocol for streaming XML in near-
   real-time that is used mainly for the purpose of instant messaging
   (IM) and presence by the servers, clients, and other applications
   that comprise the Jabber network.

Table of Contents

[1](1). Introduction

[1.1](1.1) Overview

   The Extensible Messaging and Presence Protocol (XMPP) is an open XML
   [[1](1)] protocol for near-real-time messaging, presence, and request-
   response services. The protocol was developed originally within the
   Jabber community starting in 1998, and has continued to evolve under
   the auspices of the Jabber Software Foundation [[2](2)] (since June 2001)
   and the XMPP WG (since November 2002). The current document defines
   the core features of XMPP; XMPP IM [[3](3)] defines the extensions
   necessary to provide the instant messaging (IM) and presence
   functionality defined in [RFC 2779](RFC 2779) [[4](4)].

[1.2](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](RFC 2119) [[5](5)].

[1.3](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/](http://www.jabber.org/cgi-bin/mailman/listinfo/xmppwg/)>.

[1.4](1.4) Intellectual Property Notice

   This document is in full compliance with all provisions of [Section 10
   of RFC 2026](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](link). **Generalized Architecture**

[2.1](link) **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 [[6](link)] 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:

   o C1, C2, C3 -- XMPP clients

   o S1, S2 -- XMPP servers

   o G1 -- A gateway that translates between XMPP and the protocol(s)
      used on a foreign (non-XMPP) messaging network

   o FN1 -- A foreign messaging network

   o FC1 -- A client on a foreign messaging network

[2.2](link) **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
   received from connected clients, servers, and services.

2.3 Client

   Most clients connect directly to a server over a TCP socket and use
   XMPP to take full advantage of the functionality provided by a server
   and any associated services, although it must be noted that 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).
   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 [7] for connections between a Jabber client and a Jabber
   server is 5222. For further details about client-to-server
   communications expressly for the purpose of instant messaging and
   presence, refer to XMPP IM [3].

2.4 Gateway

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

2.5 Network

   Because each server is identified by a network address (typically a
   DNS hostname) and because server-to-server communications are a
   straightforward extension of the client-to-server protocol, in
   practice the system consists of a network of servers that inter-
   communicate. Thus user-a@domain1 is able to exchange messages,
   presence, and other information with user-b@domain2. This pattern is
   familiar from messaging protocols (such as SMTP) that make use of
   network addressing standards. Upon opening a TCP socket on the IANA-
   registered port 5269, there are two methods for negotiating a
   connection between any two servers: server dialback (Section 6.2) and
   SASL authentication (Section 6.1).

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 [8]. 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 [9] and RFC 1123
[10]. A domain identifier MUST be no more than 1023 bytes in length
and MUST conform to the nameprep [11] profile of stringprep [12].

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 [13] profile of stringprep [12].

3.4 Resource Identifier

The resource identifer is an optional tertiary identifier. 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. 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 [14] profile of stringprep [12].

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:

   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. An XML stream 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. The end of the XML stream is denoted unambiguously
      be a closing XML </stream> tag.

   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 or CDATA
      sections 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
   desired to close the stream, it simply sends a closing </stream> tag
   to the server (alternatively, the session may be closed by the
   server).

   Thus a client's session with a server can be seen as two open-ended
   XML documents that are built up through the accumulation of the XML
   stanzas sent over the two XML streams (i.e., one from the client to
   the server and one from the server to the client), and the root

element can be considered the document entity for each
document. In essence, then, an XML stream acts as an envelope for
all the XML stanzas sent during a session. We can represent this
graphically as follows:

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

4.2 Restrictions

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

4.3 Stream Attributes

The attributes of the stream element are as follows:

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

o from -- The 'from' attribute SHOULD be used only in the XML stream
   from the receiving entity to the initiating entity, and MUST be

set to the XMPP address of the receiving entity granting access to
the initiating entity. There SHOULD be no 'from' attribute on the
XML stream sent from the initiating entity to the receiving
entity; however, if a 'from' attribute is included, it SHOULD be
ignored by the receiving entity.

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

o version -- The 'version' attribute MAY be used in the XML stream
   from the initiating entity to the receiving entity in order signal
   compliance with the protocol defined herein; this is done by
   setting the value of the attribute to "1.0". If the initiating
   entity includes the version attribute 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       | JID of receiver          | ignored
from     | ignored                  | JID of receiver
id       | ignored                  | session key
version  | signals XMPP 1.0 support | signals XMPP 1.0 support
```

4.4 Namespace Declarations

The stream element MAY contain namespace declarations as defined in
the XML namespaces specification [15].

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

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

prefix on the <stream/> element; however, for historical reasons some
entities MAY accept only a 'stream' prefix, resulting in the use of a
<stream:stream/> element as the stream root. The name of the stream
namespace MUST be "http://etherx.jabber.org/streams".

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, existing
implementations MAY support only these two default namespaces):

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

o 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.

4.5 Stream Features

The root stream element MAY contain a features child element (e.g.,
<stream:features/> if the stream namespace prefix is 'stream'). This
is used to communicate generic stream-level capabilities including
stream-level features that can be negotiated as the streams are set
up. If the initiating entity sends a "version='1.0'" 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 ignore them.

4.6 **Stream Errors**

   The root stream element MAY contain an error child element (e.g.,
   <stream:error/> if the stream namespace prefix is 'stream'). The
   error child MUST be sent by a Jabber entity (usually a server rather
   than a client) if it perceives that a stream-level error has
   occurred.

4.6.1 **Rules**

   The following rules apply to stream-level errors:

   o 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 and then send a closing </stream> tag.

   o 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.

4.6.2 **Syntax**

   The syntax for stream errors is as follows:

   <stream:error class='error-class'>
     <stream-condition xmlns='urn:ietf:rfc:xmppcore-rfc-number:streams'>
     </stream-condition>
   </stream:error>

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

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

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

   o redirect -- the condition relates to a host redirection

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

The <stream-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:rfc:xmppcore-rfc-number:streams' that scopes theelement adheres to the format defined in RFC 2648 [16].)

4.6.3 Conditions

The following stream-level error conditions are defined:

o -- 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".

o -- 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".

o -- 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".

o -- the stream namespace name is something
   other than "http://etherx.jabber.org/streams"; the associated
   class is "format".

o -- the server is resource-contrained and is
   unable to service the stream; the associated class is "server".

o -- 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".

o -- the server is being shut down and all active
   streams are being closed; the associated class is "server".

o -- the initiating entity has sent a
   first-level child of the stream that is not supported by the
   server; the associated class is "format".

o -- 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".

o <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".

4.6.4 Extensibility

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

4.7 Simple Streams Example

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

A basic session:

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

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

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

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

A session gone bad:

C: <?xml version='1.0'?>
   <stream:stream
       to='server'
       xmlns='jabber:client'
       xmlns:stream='http://etherx.jabber.org/streams'
       version='1.0'>
S: <?xml version='1.0'?>
   <stream:stream
       from='server'
       id='id_123456789'
       xmlns='jabber:client'
       xmlns:stream='http://etherx.jabber.org/streams'
       version='1.0'>
C: <message><body>Bad XML, no closing body tag!</message>
S: <stream:error class='client'>
     <stream-condition xmlns='urn:ietf:rfc:xmppcore-rfc-number:streams'>
      <xml-not-well-formed/>
     </stream-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) [17] protocol, along with a "STARTTLS"
   extension that is modelled on similar extensions for the IMAP [18],
   POP3 [19], and ACAP [20] protocols as described in RFC 2595 [21].
   The namespace identifier for the STARTTLS extension is 'http://
   www.ietf.org/rfc/rfc2595.txt'. TLS may 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).

   The following rules MUST be observed:

   1. If the initiating entity is capable of using the STARTTLS
       extension, it MUST include the "version='1.0'" flag in the
       initiating stream header.

   2. If the receiving entity is capable of using the STARTTLS
       extension, it MUST send the <starttls/> element in the defined
       namespace along with the list of features that it sends in
       response to the opening stream tag received from the initiating
       entity.

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

   4. If the TLS negotiation is successful, TLS takes effect
       immediately following the closing ">" character of theelement for the client and immediately following the closing
       ">" character of the <proceed> element for the server. A new
       stream MUST then be initiated by the initiating entity.

   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. Whether the TLS negotiation results in success or failure, the
    initiating entity SHOULD continue with SASL negotiation.

9. 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.

5.2 Narrative

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

1. Then 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 then initiating entity.

3. The receiving entity offers the STARTTLS extension to then
    initiating entity by sending it along with the list of supported
    stream features.

4. Then 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 an empty
    element or an empty element, but keep the underlying
    TCP connection open.

6. Then initiating entity begins a TLS negotiation in accordance
    with RFC 2246 [17]. Upon completion of the negotiation, then
    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
    then 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:

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

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

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

   Step 5: Server informs client to proceed:

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

   Step 5 (alt): Server informs client that TLS negotiation has failed
   (client SHOULD continue with stream authentication (Section 6)):

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

   Step 6: Client and server complete TLS negotiation via TCP.

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

    By bilateral agreement, server administrators SHOULD choose to use
    TLS between two domains for the purpose of securing server-to-server
    communications.

    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:

   <stream:features>
     <starttls xmlns='http://www.ietf.org/rfc/rfc2595.txt'/>
     <mechanisms xmlns='http://www.iana.org/assignments/sasl-mechanisms'>
       <mechanism>DIGEST-MD5</mechanism>
       <mechanism>KERBEROS_V4</mechanism>
     </mechanisms>
   </stream:features>

   Step 4: Server1 sends the STARTTLS command to Server2:

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

   Step 5: Server2 informs Server1 to proceed:

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

   Step 5 (alt): Server2 informs Server1 that TLS negotiation has failed
   (Server1 SHOULD continue with stream authentication (Section 6)):

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

   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='http://www.iana.org/assignments/sasl-mechanisms'>
    <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. When one entity is already known to another (i.e.,
   there is an existing trust relationship between the entities such as
   that established when a user registers with a server or an
   administrator configures a server to trust another server), the
   preferred method for authenticating streams between the two entities
   uses an XMPP adaptation of the Simple Authentication and Security
   Layer (SASL) [[22](#)]. When there is no existing trust relationship
   between two servers, 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. Both SASL authentication and dialback are described in
   this section.

   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](#)
   [[22](#)] (the namespace identifier for this protocol is 'http://
   www.iana.org/assignments/sasl-mechanisms').

   The following rules MUST be observed:

   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 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.

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 that is sent as a child
       of the first-level <features/> element. 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 [22]).

   3. The initiating entity selects a mechanism by sending an
       element 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 to the initiating
       entity; this element MAY optionally contain character data.

   5. The initiating entity responds to the challenge by sending a

<response/> element to the receiving entity; this element MAY
optionally contain character data.

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 to the initiating entity. The particular
    cause of failure optionally may be communicated in the 'code'
    attribute of the <failure/> element, and may be 432 (password
    transition is needed), 534 (authentication mechanism is too
    weak), or 454 (temporary authentication failure).

3. The receiving entity reports success of the handshake by sending
    a <success/> element 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 [22] 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 possesed 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

element.

   security layer negotiation: If a security layer is negotiated, both
      sides consider the original stream closed and new
      headers are sent by both entities. The security layer takes
      effect immediately following the ">" character of the empty
      element for the client and immediately following the
      closing ">" character of the element for the server.

   use of the authorization identity: The authorization identity, if
      present, is unused by xmpp.

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

Step 4: Client selects an authentication mechanism ("initial
response"):

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

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

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

The decoded challenge is:

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

Step 6: Client responds to the challenge:

```
<response xmlns='http://www.iana.org/assignments/sasl-mechanisms'>
    dXNlcm5hbWU9InJvYiIscmVhbG09ImNhdGFjbHlzbS5jeCIsbm9uY2U9Ik
    9BNk1HOXRFUUdtMmhoIixcIGNub25jZT0iT0E2TUhYaDZWcVRyUmsiLG5j
    PTAwMDAwMDAxLHFvcD1hdXRoLFwgZGlnZXN0LXVyaT0ieG1wcC9jYXRhY2
    x5c20uY3giLFwgcmVzcG9uc2U9ZDM4OGRhZDkwZDRiYmQ3NjBhMTUyMzIxZ
    jIxNDNhZjcsY2hhcnNldD11dGYtOA==
</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
```

Step 7: Server sends another challenge to the client:

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

The decoded challenge is:

```
rspauth=ea40f60335c427b5527b84dbabcdfffd
```

Step 8: Client responds to the challenge:

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

Step 9: Server informs client of successful authentication:

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

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

```
<failure xmlns='http://www.iana.org/assignments/sasl-mechanisms' code='454'/
>
```

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

By bilateral agreement, server administrators MAY choose to use SASL
between two domains for the purpose of securing server-to-server
communications.

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='http://www.iana.org/assignments/sasl-mechanisms'>
    <mechanism>DIGEST-MD5</mechanism>
    <mechanism>KERBEROS_V4</mechanism>
  </mechanisms>
</stream:features>
```

Step 4: Server1 selects an authentication mechanism ("initial response"):

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

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

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

The decoded challenge is:

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

Step 6: Server1 responds to the challenge:

```
<response xmlns='http://www.iana.org/assignments/sasl-mechanisms'>
    cmVhbG09ImNhdGFjbHlzbS5jeCIsbm9uY2U9Ik9BNk1HOXRFUUdtMmhoIixjbm9uY2U9B
    Nk1IWGg2VnFUclJrIixuYz0wMDAwMDAwMSxxb3A9YXV0aCxkaWdlc3QtdXJpPSJ4bXBwL2Nh
    dGFjbHlzbS5jeCIscmVzcG9uc2U9ZDM4OGRhZDkwZDRiYmQ3NjBhMTUyMzIxZjIxNDNhZjcs
    Y2hhcnNldD11dGYtOAo=
</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='http://www.iana.org/assignments/sasl-mechanisms'>
       cnNwYXV0aD1lYTQwZjYwMzM1YzQyN2I1NTI3Yjg0ZGJhYmNkZmZmZA==
   </challenge>
```

   The decoded challenge is:

```
   rspauth=ea40f60335c427b5527b84dbabcdfffd
```

   Step 8: Server1 responds to the challenge:

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

   Step 9: Server2 informs Server1 of successful authentication:

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

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

```
   <failure xmlns='http://www.iana.org/assignments/sasl-mechanisms' code='454'/
>
```

   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 (at least as much as the DNS can
   be trusted). The method is called dialback and is used only within
   XML streams that are declared under the "jabber:server" namespace.

   The purpose of the dialback protocol is to make server spoofing more
   difficult, and thus to make it more difficult to forge XML stanzas.
   Dialback is not intended as a mechanism for securing or encrypting
   the streams between servers 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. Domains requiring more robust security SHOULD use TLS
   or 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 server on the Jabber network. All
   DNS hostname resolutions MUST first resolve the hostname using an SRV
   [24] 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 [7].

   Note: the method for generating and verifying the keys used in the
   dialback protocol MUST take into account the hostnames being used,
   along with a secret known only by the receiving server and the random
   ID generated for the stream. Generating unique but verifiable keys
   is important to prevent common man-in-the-middle attacks and server
   spoofing.

   In the description that follows we use the following terminology:

   o Originating Server -- the server that is attempting to establish a
      connection between the two servers

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

   o Authoritative Server -- the server that is given when a DNS lookup
      is performed on the name that Originating Server initially gave;
      for basic environments this will be Originating Server, but it
      could be a separate machine in Originating Server's network

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

   1. Originating Server establishes a connection to Receiving Server.

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

3. Receiving Server establishes a connection to Authoritative
    Server.

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

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

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

We can represent this flow of events graphically as follows:

```
Originating                   Receiving
    Server                       Server
-----------               ---------
      |                        |
      |   establish connection |
      | ---------------------> |
      |                        |
      |    send stream header  |
      | ---------------------> |
      |                        |
      |   establish connection |
      | <--------------------- |
      |                        |
      |    send stream header  |
      | <--------------------- |
      |                        |                         Authoritative
      |    send dialback key   |                             Server
      | ---------------------> |                         -------------
      |                        |                              |
      |                        |   establish connection       |
      |                        | ---------------------->      |
      |                        |                              |
      |                        |    send stream header        |
      |                        | ---------------------->      |
      |                        |                              |
      |                        |   establish connection       |
      |                        | <----------------------      |
      |                        |                              |
      |                        |    send stream header        |
      |                        | <----------------------      |
      |                        |                              |
      |                        |    send dialback key         |
```
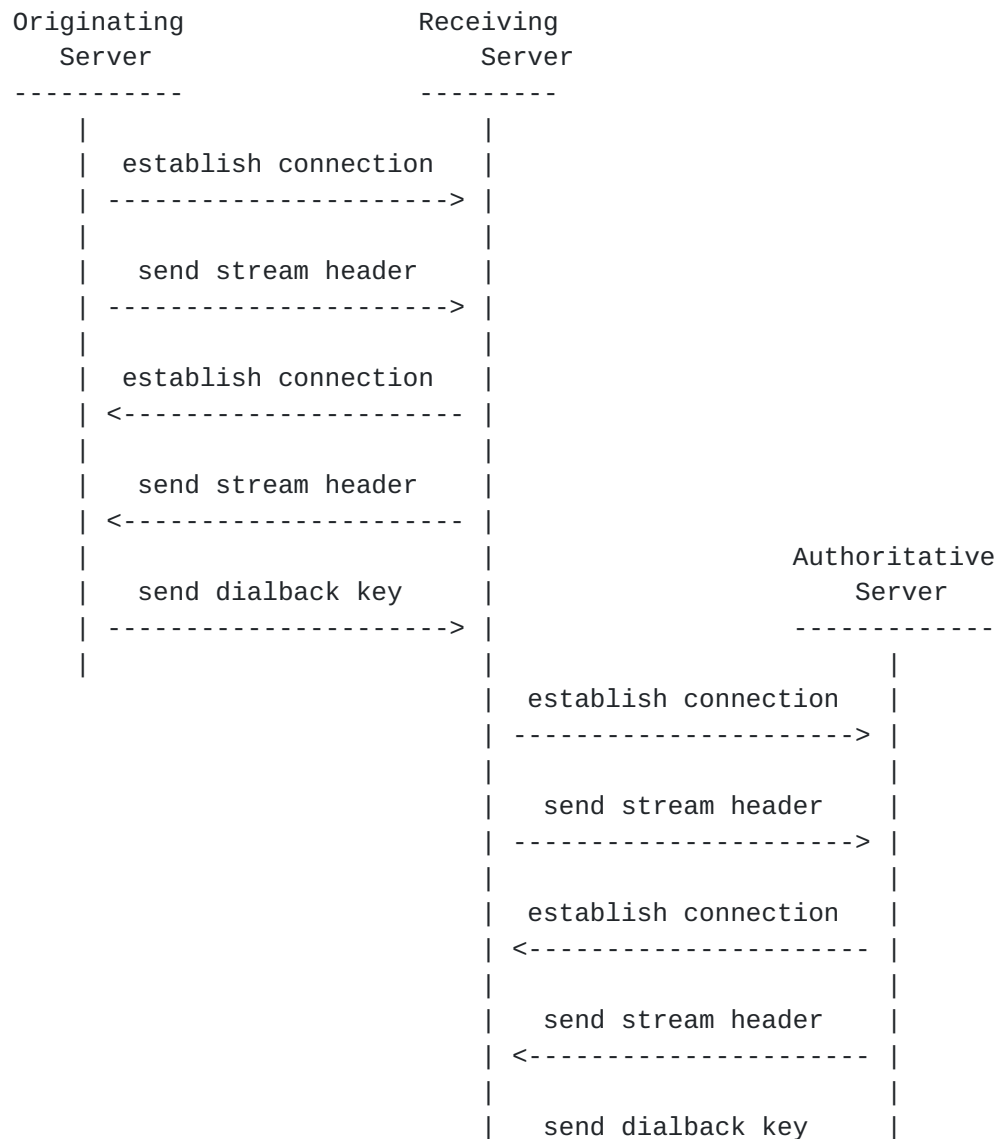
```
                                    | ---------------------> |
                                    |                        |
                                    |  validate dialback key |
                                    | <--------------------- |
                                    |
            |  report dialback result |
            | <--------------------- |
            |                        |
```

[6.2.1](#) **Dialback Protocol**

   The traffic sent between the servers is as follows:

   1.   Originating Server establishes TCP connection to Receiving
        Server

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

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

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

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

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

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

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

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

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

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

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

    7.   Authoritative Server sends Receiving Server a stream header:

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

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

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

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

    9.   Authoritative Server sends a stanza back to Receiving Server

        verifying whether the key was valid or invalid:

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

         or

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

   10. Receiving Server informs Originating Server of the result:

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

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

[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.

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

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 [23].

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:

   o chat

   o error

   o groupchat

   o headline

   For information about the meaning of these message types, refer to
   XMPP IM [3].

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", and as long as the element name
   does not match that of one of the core data elements, stream
   elements, or defined children thereof.

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

7.3.2.2 **Subject**

The <subject/> element specifies the topic of the message. The
element MUST NOT possess any attributes, with the
exception of the 'xml:lang' attribute. Multiple instances of the
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 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 element is
optional and is not used to identify individual messages, only
conversations. Only one element MAY be included in a
message stanza, and it MUST NOT possess any attributes. Theelement 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@host/resource) with the
    recipient's full JID

2. concatenate these JID strings with a full ISO-8601 timestamp
    including year, month, day, hours, minutes, seconds, and UTC
    offset 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:

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

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

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

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

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

   o   probe -- A request for an entity's current presence. In general
       SHOULD NOT be sent by a client.

   o   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 [3].

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", and as long as the element name
   does not match that of one of the core data elements, stream
   elements, or defined children thereof.

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

o   away

o   chat

o   xa

o   dnd

For information about the meaning of these values, refer to XMPP IM
[3].

7.4.2.2 Status

The optional <status/> element specifies 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
element MUST NOT possess any attributes, with the exception of the
'xml:lang' attribute. Multiple instances of the 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 element specifies the priority level of the
connected resource. The value may be any integer between -128 to
127. Only one 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 [3].

7.5 IQ Stanzas

7.5.1 Overview

   Info/Query, or IQ, is a request-response mechanism, similar in some
   ways to HTTP [25]. 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):

```
   Requesting                   Responding
     Entity                       Entity
   ----------                   ----------
       |                            |
       | <iq type='get' id='1'>     |
       | ----------------------->   |
       |                            |
       | <iq type='result' id='1'>  |
       | <-----------------------   |
       |                            |
       | <iq type='set' id='2'>     |
       | ----------------------->   |
       |                            |
       | <iq type='result' id='2'>  |
       | <-----------------------   |
       |                            |
```

   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 MAY be ignored):

o get -- The stanza is a request for information.

o set -- The stanza provides required data, sets new values, or
   replaces existing values.

o result -- The stanza is a response to a successful get or set
   request.

o 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", and as long as the element name does not
match that of one of the core data elements, stream elements, or
defined children thereof. 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.

If the IQ stanza is of type "error", it MUST include an
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 any element (other than the core data elements, stream
elements, or defined children thereof). The child element MUST
possess an 'xmlns' namespace declaration (other than the stream
namespace and the default namespace) that defines all data contained
within the child element.

Support for any given extended namespace is OPTIONAL on the part of
any implementation. If an entity does not understand such a
namespace, it MUST ignore the associated XML data (if the stanza is
being routed on to another entity, ignore means "pass it on

untouched"). If an entity receives 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/>.

7.7 Stanza Errors

As defined below, stanza-related errors are handled in a manner
similar to stream errors (Section 4.6).

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

o 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.

o An 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'>
    <stanza-condition xmlns='urn:ietf:rfc:xmppcore-rfc-number:stanzas'>
      <descriptive-element-name/>
    </stanza-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:

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

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

o app -- the condition is particular to an application and is
   specified in a namespace other than 'urn:ietf:rfc:xmppcore-rfc-
   number:stanzas'

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

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

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

The <stanza-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:rfc:xmppcore-rfc-number:stanzas' that scopes theelement adheres to the format defined in RFC 2648 [16].)

7.7.3 Conditions

The following stanza-related error conditions are defined:

o -- 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".

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

o -- the stanza is understood but the action is
   forbidden; the associated class is "access".

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

o -- 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".

o <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".

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

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

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

o <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".

o <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".

o <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;
this MUST be contained in a properly-namespaced child of theelement (i.e., the namespace name MUST NOT be one of
namespace names defined herein).

8. **XML Usage within XMPP**

8.1 **Namespaces**

   XML Namespaces [15] 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.

8.2 **Validation**

   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.

8.3 **Character Encodings**

   Software implementing XML streams MUST support the UTF-8 (RFC 2279
   [26]) and UTF-16 (RFC 2781 [27]) transformations of Universal
   Character Set (ISO/IEC 10646-1 [28]) characters. Software MUST NOT
   attempt to use any other encoding for transmitted data. The
   encodings of the transmitted and received streams are independent.
   Software MAY select either UTF-8 or UTF-16 for the transmitted
   stream, and SHOULD deduce the encoding of the received stream as
   described in the XML specification [1]. For historical reasons,
   existing implementations MAY support UTF-8 only.

8.4 **Inclusion of Text Declaration**

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

9. IANA Considerations

The IANA registers "xmpp" as a GSSAPI [30] 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, the 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 must be established out-
   of-band (i.e., no self-signed certificates).

   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 [29].

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.

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 [7]. 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)

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]   Jabber Software Foundation, "Jabber Software Foundation",
          August 2001, <http://www.jabber.org/>.

   [3]   Saint-Andre, P. and J. Miller, "XMPP Instant Messaging (draft-
          ietf-xmpp-im-06, work in progress)", March 2003.

   [4]   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>.

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

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

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

   [8]   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>.

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

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

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

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

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

   [14]   Saint-Andre, P. and J. Hildebrand, "Resourceprep: A Stringprep

Profile for Resource Identifiers in XMPP (draft-ietf-xmpp-resourceprep-01, work in progress)", February 2003.

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

[16]   Moats, R., "A URN Namespace for IETF Documents", RFC 2648, August 1999.

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

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

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

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

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

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

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

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

[25]   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.

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

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

[28]   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.

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

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

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='http://www.ietf.org/rfc/rfc2595.txt'
      xmlns='http://www.ietf.org/rfc/rfc2595.txt'
      elementFormDefault='qualified'>

    <xs:element name='starttls'/>
    <xs:element name='proceed'/>
    <xs:element name='failure'/>

  </xs:schema>
```

A.3 SASL namespace

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

<xs:schema
    xmlns:xs='http://www.w3.org/2001/XMLSchema'
    targetNamespace='http://www.iana.org/assignments/sasl-mechanisms'
    xmlns='http://www.iana.org/assignments/sasl-mechanisms'
    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:attribute name='code' type='xs:string' use='optional'/>
    </xs:complexType>
  </xs:element>

</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='xs:NMTOKEN' 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='xs:NMTOKEN' use='optional'/>
          </xs:complexType>
        </xs:element>

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

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

    <xs:element name='iq'>
      <xs:complexType>
        <xs:choice maxOccurs='unbounded'>
```

```
            <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='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:schema>
```

```
    <?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='xs:NMTOKEN' 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='xs:NMTOKEN' use='optional'/>
    </xs:complexType>
  </xs:element>

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

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

```
   <xs:element name='iq'>
     <xs:complexType>
       <xs:choice maxOccurs='unbounded'>
         <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='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: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:rfc:xmppcore-rfc-number:streams'
    xmlns='urn:ietf:rfc:xmppcore-rfc-number:streams'
    elementFormDefault='qualified'>

  <xs:element name='stream-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-namespace'/>
        <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' type='xs:string'/>
  <xs:element name='host-unknown' type='xs:string'/>
  <xs:element name='internal-server-error' type='xs:string'/>
  <xs:element name='invalid-namespace' type='xs:string'/>
  <xs:element name='resource-constraint' type='xs:string'/>
  <xs:element name='see-other-host' type='xs:string'/>
  <xs:element name='system-shutdown' type='xs:string'/>
  <xs:element name='unsupported-stanza-type' type='xs:string'/>
  <xs:element name='unsupported-version' type='xs:string'/>
  <xs:element name='xml-not-well-formed' type='xs:string'/>

</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:rfc:xmppcore-rfc-number:stanzas'
    xmlns='urn:ietf:rfc:xmppcore-rfc-number:stanzas'
```

```
      elementFormDefault='qualified'>

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

</xs:schema>
```

Appendix B. Provisional Namespace Names

Note to RFC editor: prior to publication, the string 'xmppcore-rfc-number' must be replaced in all instances by the RFC number assigned to this draft. (In addition, please remove this appendix, and the corresponding entry in the table of contents, prior to publication.)

Appendix C. Revision History

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

C.1 Changes from **draft-ietf-xmpp-core-05**

   o Added <invalid-namespace/> as a stream error condition.

   o Adjusted security considerations per discussion at IETF 56 and on
      list.

C.2 Changes from **draft-ietf-xmpp-core-04**

   o Added server-to-server examples for TLS and SASL.

   o Changed error syntax, rules, and examples based on list
      discussion.

   o Added schemas for the TLS, stream error, and stanza error
      namespaces.

   o Added note to RFC editor regarding provisional namespace names.

   o Made numerous small editorial changes and clarified text
      throughout.

C.3 Changes from **draft-ietf-xmpp-core-03**

   o Clarified rules and procedures for TLS and SASL.

   o Amplified stream error code syntax per list discussion.

   o Made numerous small editorial changes.

C.4 Changes from **draft-ietf-xmpp-core-02**

   o Added dialback schema.

   o Removed all DTDs since schemas provide more complete definitions.

   o Added stream error codes.

   o Clarified error code "philosophy".

C.5 **Changes from draft-ietf-xmpp-core-01**

o Updated the addressing restrictions per list discussion and added
   references to the new nodeprep and resourceprep profiles.

o Corrected error in Stream Authentication regarding "version='1.0'"
   flag.

o Made numerous small editorial changes.

C.6 **Changes from draft-ietf-xmpp-core-00**

o Added information about TLS from list discussion.

o Clarified meaning of "ignore" based on list discussion.

o Clarified information about Universal Character Set data and
   character encodings.

o Provided base64-decoded information for examples.

o Fixed several errors in the schemas.

o Made numerous small editorial fixes.

C.7 **Changes from draft-miller-xmpp-core-02**

o Brought Streams Authentication section into line with discussion
   on list and at IETF 55 meeting.

o Added information about the optional 'xml:lang' attribute per
   discussion on list and at IETF 55 meeting.

o Specified that validation is neither required nor recommended, and
   that the formal definitions (DTDs and schemas) are included for
   descriptive purposes only.

o Specified that the response to an IQ stanza of type 'get' or 'set'
   must be an IQ stanza of type 'result' or 'error'.

o Specified that compliant server implementations must process
   stanzas in order.

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

o 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.

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

o Removed references to TLS pending list discussion.

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

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

o Noted that XMPP activity respects firewall administration
   policies.

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

o 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).