          Extensible Messaging and Presence Protocol (XMPP): Core

Status of this Memo

Copyright Notice

Abstract

   This memo defines the core features of the Extensible Messaging and
   Presence Protocol (XMPP), a protocol for streaming Extensible Markup
   Language (XML) elements in order to exchange structured information
   in close to real time between any two network endpoints. While XMPP
   provides a generalized, extensible framework for exchanging XML data,
   it is used mainly for the purpose of building instant messaging and
   presence applications that meet the requirements of RFC 2779.

Table of Contents

1. Introduction

1.1. Overview

   The Extensible Messaging and Presence Protocol (XMPP) is an open
   Extensible Markup Language [XML] protocol for near-real-time
   messaging, presence, and request-response services. The basic syntax
   and semantics were developed originally within the Jabber open-source
   community, mainly in 1999. In 2002, the XMPP WG was chartered with
   developing an adaptation of the Jabber protocol that would be
   suitable as an IETF instant messaging (IM) and presence technology.
   As a result of work by the XMPP WG, the current memo defines the core
   features of XMPP 1.0; the extensions required to provide the instant
   messaging and presence functionality defined in RFC 2779 [IMP-REQS]
   are specified in the Extensible Messaging and Presence Protocol
   (XMPP): Instant Messaging and Presence [XMPP-IM].

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 BCP
   14, RFC 2119 [TERMS].

2. Generalized Architecture

2.1. Overview

   Although XMPP is not wedded to any specific network architecture, to
   date it usually has been implemented via a client-server architecture
   wherein a client utilizing XMPP accesses a server over a [TCP]
   connection, and servers also communicate with each other over TCP
   connections.

   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. Server

   A server acts as an intelligent abstraction layer for XMPP
   communications. Its primary responsibilities are:

   o to manage connections from or sessions for other entities, in the
      form of XML streams (Section 4) to and from authorized clients,
      servers, and other entities

o to route appropriately-addressed XML stanzas (Section 9) 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 instant messaging and presence 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.

2.3. Client

Most clients connect directly to a server over a [TCP] connection and
use XMPP to take full advantage of the functionality provided by a
server and any associated services. Multiple resources (e.g.,
devices or locations) MAY connect simultaneously to a server on
behalf of each authorized client, with each resource differentiated
by the resource identifier of an XMPP address (e.g., <node@domain/
home> vs. <node@domain/work>) as defined under Addressing Scheme
(Section 3). The RECOMMENDED port for connections between a client
and a server is 5222, as registered with the IANA (see Port Numbers
(Section 15.9)).

2.4. Gateway

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

2.5. Network

Because each server is identified by a network address 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, for example,
<juliet@example.com> is able to exchange messages, presence, and
other information with <romeo@example.net>. This pattern is familiar
from messaging protocols (such as [SMTP]) that make use of network
addressing standards. Communications between any two servers are
OPTIONAL. If enabled, such communications SHOULD occur over XML
streams that are bound to [TCP] connections. The RECOMMENDED port
for connections between servers is 5269, as registered with the IANA
(see Port Numbers (Section 15.9)).

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 [URI]. For historical reasons, the address
   of an XMPP entity is called a Jabber Identifier or JID. A valid JID
   contains a set of ordered elements formed of a domain identifier,
   node identifier, and resource identifier.

   The syntax for a JID is defined below using the Augmented Backus-Naur
   Form as defined in [ABNF]. (The IPv4address and IPv6address rules
   are defined in Appendix B of [IPv6]; the allowable character
   sequences that conform to the node rule are defined by the Nodeprep
   profile of [STRINGPREP] as documented in Appendix A of this memo; the
   allowable character sequences that conform to the resource rule are
   defined by the Resourceprep profile of [STRINGPREP] as documented in
   Appendix B of this memo; and the sub-domain rule makes reference to
   the concept of an internationalized domain label as described in
   [IDNA].)

      jid   = [node "@"] domain ["/" resource]
      domain   = fqdn / address-literal
      fqdn   = (sub-domain 1*("." sub-domain))
      sub-domain   = (internationalized domain label)
      address-literal = IPv4address / IPv6address

   All JIDs are based on the foregoing structure. The most common use
   of this structure is to identify an instant messaging user, the
   server to which the user connects, and the user's connected resource
   (e.g., a specific client) in the form of <user@host/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).

   Each allowable portion of a JID (node identifier, domain identifier,
   and resource identifier) MUST NOT be more than 1023 bytes in length,
   resulting in a maximum total size (including the '@' and '/'
   separators) of 3071 bytes.

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 that provides functionality above and beyond
   the capabilities of a server (e.g., a multi-user chat service, a user
   directory, or a gateway to a foreign messaging system).

   The domain identifier for every server or service that will
   communicate over a network MAY be an IP address but SHOULD be a fully
   qualified domain name (see [DNS]). A domain identifier MUST be an
   "internationalized domain name" as defined in [IDNA], to which the
   Nameprep [NAMEPREP] profile of stringprep [STRINGPREP] can be applied
   without failing. Before comparing two domain identifiers, a server
   MUST (and a client SHOULD) first apply the Nameprep profile to the
   labels (as defined in [IDNA]) that make up each identifier.

3.3. Node Identifier

   The node identifier is an optional secondary identifier placed before
   the domain identifier and separated from the latter by the '@'
   character. 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 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 instant messaging and presence applications
   of XMPP, this address is called a "bare JID" and is of the form
   <node@domain>.

   A node identifier MUST be formatted such that the Nodeprep profile of
   [STRINGPREP] can be applied to it without failing. Before comparing
   two node identifiers, a server MUST (and a client SHOULD) first apply
   the Nodeprep profile to each identifier.

3.4. Resource Identifier

   The resource identifier is an optional tertiary identifier placed
   after the domain identifier and separated from the latter by the '/'
   character. A resource identifier may modify either a <node@domain>
   or a mere <domain> address. It usually represents a specific
   session, connection (e.g., a device or location), or object (e.g., a
   participant in a multi-user chat room) belonging to the entity
   associated with a node identifier. A resource identifier is opaque

to both servers and other clients, and is typically defined by a
client implementation when it provides the information necessary to
complete Resource Binding (Section 7) (although it may be generated
by a server on behalf of a client), after which it is referred to as
a "connected resource". An entity MAY maintain multiple connected
resources simultaneously, with each connected resource differentiated
by a distinct resource identifier.

A resource identifier MUST be formatted such that the Resourceprep
profile of [STRINGPREP] can be applied without failing. Before
comparing two resource identifiers, a server MUST (and a client
SHOULD) first apply the Resourceprep profile to each identifier.

3.5. Determination of Addresses

After SASL negotiation (Section 6) and, if appropriate, Resource
Binding (Section 7), the receiving entity for a stream MUST determine
the initiating entity's JID.

For server-to-server communications, the initiating entity's JID
SHOULD be the authorization identity, derived from the authentication
identity, as defined by the Simple Authentication and Security Layer
(SASL) specification [SASL], if no authorization identity was
specified during SASL negotiation (Section 6).

For client-to-server communications, the "bare JID" (<node@domain>)
SHOULD be the authorization identity, derived from the authentication
identity, as defined in [SASL], if no authorization identity was
specified during SASL negotiation (Section 6); the resource
identifier portion of the "full JID" (<node@domain/resource>) SHOULD
be the resource identifier negotiated by the client and server during
Resource Binding (Section 7).

The receiving entity MUST ensure that the resulting JID (including
node identifier, domain identifier, resource identifier, and
separator characters) conforms to the rules and formats defined
earlier in this section; to meet this restriction, the receiving
entity may need to replace the JID sent by the initiating entity with
the canonicalized JID as determined by the receiving entity.

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. These
terms are defined as follows:

   Definition of XML Stream: An XML stream is a container for the
      exchange of XML elements between any two entities over a network.
      The start of an XML stream is denoted unambiguously by an opening
      XML <stream> tag (with appropriate attributes and namespace
      declarations), while the end of the XML stream is denoted
      unambiguously by a closing XML </stream> tag. During the life of
      the stream, the entity that initiated it can send an unbounded
      number of XML elements over the stream, either elements used to
      negotiate the stream (e.g., to negotiate Use of TLS (Section 5) or
      use of SASL (Section 6)) or XML stanzas (as defined herein,
      , , or elements qualified by the
      default namespace). The "initial stream" is negotiated from the
      initiating entity (usually a client or server) to the receiving
      entity (usually a server), and can be seen as corresponding to the
      initiating entity's "session" with the receiving entity. The
      initial stream enables unidirectional communication from the
      initiating entity to the receiving entity; in order to enable
      information exchange from the receiving entity to the initiating
      entity, the receiving entity MUST negotiate a stream in the
      opposite direction (the "response stream").

   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 the production [43] content of [XML].
      The start of any XML stanza is denoted unambiguously by the
      element start tag at depth=1 of the XML stream (e.g., <presence>),
      and the end of any XML stanza is denoted unambiguously by the
      corresponding close tag at depth=1 (e.g., </presence>). An XML
      stanza MAY contain child elements (with accompanying attributes,
      elements, and XML character data) as necessary in order to convey
      the desired information. The only XML stanzas defined herein are
      the <message/>, <presence/>, and <iq/> elements qualified by the
      default namespace for the stream, as described under XML Stanzas
      (Section 9); an XML element sent for the purpose of Transport
      Layer Security (TLS) negotiation (Section 5), Simple
      Authentication and Security Layer (SASL) negotiation (Section 6),
      or server dialback (Section 8) is not considered to be an XML
      stanza.

   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 and the character
   encoding supported (see Inclusion of Text Declaration (Section 11.4);
   see also Character Encoding (Section 11.5)). Subject to local
   policies and service provisioning, 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 completed SASL negotiation
(Section 6), the client MAY send an unbounded number of XML stanzas
over the stream to any recipient on the network. When the client
desires to close the stream, it simply sends a closing </stream> tag
to the server (alternatively, the stream may be closed by the
server), after which both the client and server SHOULD terminate the
underlying connection (usually a TCP connection) as well.

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

In essence, then, an XML stream acts as an envelope for all the XML
stanzas sent during a session. We can represent this in a simplistic
fashion as follows:

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

4.2. Binding to TCP

   Although there is no necessary coupling of an XML stream to a [TCP]
   connection (e.g., two entities could connect to each other via
   another mechanism such as polling over [HTTP]), this specification
   defines a binding of XMPP to TCP only. In the context of
   client-to-server communications, a server MUST allow a client to
   share a single TCP connection for XML stanzas sent from client to
   server and from server to client. In the context of server-to-server
   communications, a server MUST use one TCP connection for XML stanzas
   sent from the server to the peer and another TCP connection
   (initiated by the peer) for stanzas from the peer to the server, for
   a total of two TCP connections.

4.3. Stream Security

   When negotiating XML streams in XMPP 1.0, TLS SHOULD be used as
   defined under Use of TLS (Section 5) and SASL MUST be used as defined
   under Use of SASL (Section 6). The "initial stream" (i.e., the
   stream from the initiating entity to the receiving entity) and the
   "response stream" (i.e., the stream from the receiving entity to the
   initiating entity) MUST be secured separately, although security in
   both directions MAY be established via mechanisms that provide mutual
   authentication. An entity SHOULD NOT attempt to send XML Stanzas
   (Section 9) over the stream before the stream has been authenticated,
   but if it does, then the other entity MUST NOT accept such stanzas
   and SHOULD return a <not-authorized/> stream error and then terminate
   both the XML stream and the underlying TCP connection; note well that
   this applies to XML stanzas only (i.e., <message/>, <presence/>, and
   elements scoped by the default namespace) and not to XML
   elements used for stream negotiation (e.g., elements used to
   negotiate Use of TLS (Section 5) or Use of SASL (Section 6)).

4.4. Stream Attributes

   The attributes of the stream element are as follows:

   o to -- The 'to' attribute SHOULD be used only in the XML stream
      header from the initiating entity to the receiving entity, and
      MUST be set to a hostname serviced by the receiving entity. There
      SHOULD NOT be a 'to' attribute set in the XML stream header by
      which the receiving entity replies to the initiating entity;
      however, if a 'to' attribute is included, it SHOULD be silently
      ignored by the initiating entity.

o from -- The 'from' attribute SHOULD be used only in the XML stream
   header from the receiving entity to the initiating entity, and
   MUST be set to a hostname serviced by the receiving entity that is
   granting access to the initiating entity. There SHOULD NOT be a
   'from' attribute on the XML stream header sent from the initiating
   entity to the receiving entity; however, if a 'from' attribute is
   included, it SHOULD be silently ignored by the receiving entity.

o id -- The 'id' attribute SHOULD be used only in the XML stream
   header from the receiving entity to the initiating entity. This
   attribute is a unique identifier created by the receiving entity
   to function as a session key for the initiating entity's streams
   with the receiving entity, and MUST be unique within the receiving
   application (normally a server). Note well that the stream ID may
   be security-critical and therefore MUST be both unpredictable and
   nonrepeating (see [RANDOM] for recommendations regarding
   randomness for security purposes). There SHOULD NOT be an 'id'
   attribute on the XML stream header sent from the initiating entity
   to the receiving entity; however, if an 'id' attribute is
   included, it SHOULD be silently ignored by the receiving entity.

o xml:lang -- An 'xml:lang' attribute (as defined in Section 2.12 of
   [XML]) SHOULD be included by the initiating entity on the header
   for the initial stream to specify the default language of any
   human-readable XML character data it sends over that stream. If
   the attribute is included, the receiving entity SHOULD remember
   that value as the default for both the initial stream and the
   response stream; if the attribute is not included, the receiving
   entity SHOULD use a configurable default value for both streams,
   which it MUST communicate in the header for the response stream.
   For all stanzas sent over the initial stream, if the initiating
   entity does not include an 'xml:lang' attribute, the receiving
   entity SHOULD apply the default value; if the initiating entity
   does include an 'xml:lang' attribute, the receiving entity MUST
   NOT modify or delete it (see also xml:lang (Section 9.1.5)). The
   value of the 'xml:lang' attribute MUST be an NMTOKEN (as defined
   in Section 2.3 of [XML]) and MUST conform to the format defined in
   RFC 3066 [LANGTAGS].

o version -- The presence of the version attribute set to a value of
   at least "1.0" signals support for the stream-related protocols
   (including stream features) defined in this specification.
   Detailed rules regarding the generation and handling of this
   attribute are defined below.

We can summarize as follows:

```
          | initiating to receiving | receiving to initiating
----------+-------------------------+-----------------------
to        | hostname of receiver    | silently ignored
from      | silently ignored        | hostname of receiver
id        | silently ignored        | session key
xml:lang  | default language        | default language
version   | signals XMPP 1.0 support | signals XMPP 1.0 support
```

4.4.1. Version Support

   The version of XMPP specified herein is "1.0"; in particular, this
   encapsulates the stream-related protocols (Use of TLS (Section 5),
   Use of SASL (Section 6), and Stream Errors (Section 4.7)), as well as
   the semantics of the three defined XML stanza types (<message/>,
   , and). The numbering scheme for XMPP versions is
   "<major>.<minor>". The major and minor numbers MUST be treated as
   separate integers and each number MAY be incremented higher than a
   single digit. Thus, "XMPP 2.4" would be a lower version than "XMPP
   2.13", which in turn would be lower than "XMPP 12.3". Leading zeros
   (e.g., "XMPP 6.01") MUST be ignored by recipients and MUST NOT be
   sent.

   The major version number should be incremented only if the stream and
   stanza formats or required actions have changed so dramatically that
   an older version entity would not be able to interoperate with a
   newer version entity if it simply ignored the elements and attributes
   it did not understand and took the actions specified in the older
   specification. The minor version number indicates new capabilities,
   and MUST be ignored by an entity with a smaller minor version number,
   but used for informational purposes by the entity with the larger
   minor version number. For example, a minor version number might
   indicate the ability to process a newly defined value of the 'type'
   attribute for message, presence, or IQ stanzas; the entity with the
   larger minor version number would simply note that its correspondent
   would not be able to understand that value of the 'type' attribute
   and therefore would not send it.

   The following rules apply to the generation and handling of the
   'version' attribute within stream headers by implementations:

   1. The initiating entity MUST set the value of the 'version'
       attribute on the initial stream header to the highest version
       number it supports (e.g., if the highest version number it
       supports is that defined in this specification, it MUST set the
       value to "1.0").

   2. The receiving entity MUST set the value of the 'version'
       attribute on the response stream header to either the value
       supplied by the initiating entity or the highest version number
       supported by the receiving entity, whichever is lower. The
       receiving entity MUST perform a numeric comparison on the major
       and minor version numbers, not a string match on
       "<major>.<minor>".

   3. If the version number included in the response stream header is
       at least one major version lower than the version number included
       in the initial stream header and newer version entities cannot
       interoperate with older version entities as described above, the
       initiating entity SHOULD generate an
       stream error and terminate the XML stream and underlying TCP
       connection.

   4. If either entity receives a stream header with no 'version'
       attribute, the entity MUST consider the version supported by the
       other entity to be "0.0" and SHOULD NOT include a 'version'
       attribute in the stream header it sends in reply.

4.5. Namespace Declarations

   The stream element MUST possess both a streams namespace declaration
   and a default namespace declaration (as "namespace declaration" is
   defined in the XML namespaces specification [XML-NAMES]). For
   detailed information regarding the streams namespace and default
   namespace, see Namespace Names and Prefixes (Section 11.2).

4.6. Stream Features

   If the initiating entity includes the 'version' attribute set to a
   value of at least "1.0" in the initial stream header, the receiving
   entity MUST send a <features/> child element (prefixed by the streams
   namespace prefix) to the initiating entity in order to announce any
   stream-level features that can be negotiated (or capabilities that
   otherwise need to be advertised). Currently, this is used only to
   advertise Use of TLS (Section 5), Use of SASL (Section 6), and
   Resource Binding (Section 7) as defined herein, and for Session
   Establishment as defined in [XMPP-IM]; however, the stream features
   functionality could be used to advertise other negotiable features in
   the future. If an entity does not understand or support some
   features, it SHOULD silently ignore them. If one or more security
   features (e.g., TLS and SASL) need to be successfully negotiated
   before a non-security-related feature (e.g., Resource Binding) can be
   offered, the non-security-related feature SHOULD NOT be included in
   the stream features that are advertised before the relevant security
   features have been negotiated.

4.7. Stream Errors

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

4.7.1. 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, send a closing </stream> tag, and terminate the
      underlying TCP connection.

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

4.7.2. Syntax

   The syntax for stream errors is as follows:

   <stream:error>
     <defined-condition xmlns='urn:ietf:params:xml:ns:xmpp-streams'/>
     <text xmlns='urn:ietf:params:xml:ns:xmpp-streams'
           xml:lang='langcode'>
       OPTIONAL descriptive text
     </text>
     [OPTIONAL application-specific condition element]
   </stream:error>

   The <error/> element:

   o MUST contain a child element corresponding to one of the defined
      stanza error conditions defined below; this element MUST be
      qualified by the 'urn:ietf:params:xml:ns:xmpp-streams' namespace

   o MAY contain a <text/> child containing XML character data that
      describes the error in more detail; this element MUST be qualified
      by the 'urn:ietf:params:xml:ns:xmpp-streams' namespace and SHOULD
      possess an 'xml:lang' attribute specifying the natural language of
      the XML character data

   o MAY contain a child element for an application-specific error
      condition; this element MUST be qualified by an
      application-defined namespace, and its structure is defined by
      that namespace

   The <text/> element is OPTIONAL. If included, it SHOULD be used only
   to provide descriptive or diagnostic information that supplements the
   meaning of a defined condition or application-specific condition. It
   SHOULD NOT be interpreted programmatically by an application. It
   SHOULD NOT be used as the error message presented to a user, but MAY
   be shown in addition to the error message associated with the
   included condition element (or elements).

4.7.3. Defined Conditions

   The following stream-level error conditions are defined:

   o <bad-format/> -- the entity has sent XML that cannot be processed;
      this error MAY be used instead of the more specific XML-related
      errors, such as <bad-namespace-prefix/>, <invalid-xml/>,
      , , and
      , although the more specific errors are
      preferred.

   o -- the entity has sent a namespace prefix
      that is unsupported, or has sent no namespace prefix on an element
      that requires such a prefix (see XML Namespace Names and Prefixes
      (Section 11.2)).

   o -- the server is closing the active stream for this
      entity because a new stream has been initiated that conflicts with
      the existing stream.

   o -- the entity has not generated any traffic
      over the stream for some period of time (configurable according to
      a local service policy).

   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.

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

   o <improper-addressing/> -- a stanza sent between two servers lacks
      a 'to' or 'from' attribute (or the attribute has no value).

   o <internal-server-error/> -- the server has experienced a
      misconfiguration or an otherwise-undefined internal error that
      prevents it from servicing the stream.

   o <invalid-from/> -- the JID or hostname provided in a 'from'
      address does not match an authorized JID or validated domain
      negotiated between servers via SASL or dialback, or between a
      client and a server via authentication and resource binding.

   o <invalid-id/> -- the stream ID or dialback ID is invalid or does
      not match an ID previously provided.

   o <invalid-namespace/> -- the streams namespace name is something
      other than "http://etherx.jabber.org/streams" or the dialback
      namespace name is something other than "jabber:server:dialback"
      (see XML Namespace Names and Prefixes (Section 11.2)).

   o <invalid-xml/> -- the entity has sent invalid XML over the stream
      to a server that performs validation (see Validation (Section
      11.3)).

   o <not-authorized/> -- the entity has attempted to send data before
      the stream has been authenticated, or otherwise is not authorized
      to perform an action related to stream negotiation; the receiving
      entity MUST NOT process the offending stanza before sending the
      stream error.

   o <policy-violation/> -- the entity has violated some local service
      policy; the server MAY choose to specify the policy in the <text/>
      element or an application-specific condition element.

   o <remote-connection-failed/> -- the server is unable to properly
      connect to a remote entity that is required for authentication or
      authorization.

   o <resource-constraint/> -- the server lacks the system resources
      necessary to service the stream.

   o <restricted-xml/> -- the entity has attempted to send restricted
      XML features such as a comment, processing instruction, DTD,
      entity reference, or unescaped character (see Restrictions
      (Section 11.1)).

   o <see-other-host/> -- the server will not provide service to the
      initiating entity but is redirecting traffic to another host; the
      server SHOULD specify the alternate hostname or IP address (which
      MUST be a valid domain identifier) as the XML character data of
      the <see-other-host/> element.

   o <system-shutdown/> -- the server is being shut down and all active
      streams are being closed.

   o <undefined-condition/> -- the error condition is not one of those
      defined by the other conditions in this list; this error condition
      SHOULD be used only in conjunction with an application-specific
      condition.

   o <unsupported-encoding/> -- the initiating entity has encoded the
      stream in an encoding that is not supported by the server (see
      Character Encoding (Section 11.5)).

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

   o <unsupported-version/> -- the value of the 'version' attribute
      provided by the initiating entity in the stream header specifies a
      version of XMPP that is not supported by the server; the server
      MAY specify the version(s) it supports in the <text/> element.

   o <xml-not-well-formed/> -- the initiating entity has sent XML that
      is not well-formed as defined by [XML].

4.7.4. Application-Specific Conditions

   As noted, an application MAY provide application-specific stream
   error information by including a properly-namespaced child in the
   error element. The application-specific element SHOULD supplement or
   further qualify a defined element. Thus the <error/> element will
   contain two or three child elements:

```
   <stream:error>
     <xml-not-well-formed
         xmlns='urn:ietf:params:xml:ns:xmpp-streams'/>
     <text xml:lang='en' xmlns='urn:ietf:params:xml:ns:xmpp-streams'>
       Some special application diagnostic information!
     </text>
     <escape-your-data xmlns='application-ns'/>
   </stream:error>
   </stream:stream>
```

4.8. Simplified Stream Examples

   This section contains two simplified examples of 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); these examples are included for the purpose of
   illustrating the concepts introduced thus far.

   A basic "session":

```
   C: <?xml version='1.0'?>
      <stream:stream
          to='example.com'
          xmlns='jabber:client'
          xmlns:stream='http://etherx.jabber.org/streams'
          version='1.0'>
   S: <?xml version='1.0'?>
      <stream:stream
          from='example.com'
          id='someid'
          xmlns='jabber:client'
          xmlns:stream='http://etherx.jabber.org/streams'
          version='1.0'>
   ...  encryption, authentication, and resource binding ...
   C:     <message from='juliet@example.com'
                 to='romeo@example.net'
                 xml:lang='en'>
   C:       <body>Art thou not Romeo, and a Montague?</body>
   C:     </message>
   S:     <message from='romeo@example.net'
                 to='juliet@example.com'
                 xml:lang='en'>
   S:       <body>Neither, fair saint, if either thee dislike.</body>
   S:     </message>
   C: </stream:stream>
   S: </stream:stream>
```

A "session" gone bad:

```
C: <?xml version='1.0'?>
   <stream:stream
       to='example.com'
       xmlns='jabber:client'
       xmlns:stream='http://etherx.jabber.org/streams'
       version='1.0'>
S: <?xml version='1.0'?>
   <stream:stream
       from='example.com'
       id='someid'
       xmlns='jabber:client'
       xmlns:stream='http://etherx.jabber.org/streams'
       version='1.0'>
... encryption, authentication, and resource binding ...
C: <message xml:lang='en'>
     <body>Bad XML, no closing body tag!
   </message>
S: <stream:error>
    <xml-not-well-formed
        xmlns='urn:ietf:params:xml:ns:xmpp-streams'/>
   </stream:error>
S: </stream:stream>
```

5. Use of TLS

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) protocol [TLS], along with a
   "STARTTLS" extension that is modelled after similar extensions for
   the IMAP [IMAP], POP3 [POP3], and ACAP [ACAP] protocols as described
   in RFC 2595 [USINGTLS]. The namespace name for the STARTTLS
   extension is 'urn:ietf:params:xml:ns:xmpp-tls'.

   An administrator of a given domain MAY require the use of TLS for
   client-to-server communications, server-to-server communications, or
   both. Clients SHOULD use TLS to secure the streams prior to
   attempting the completion of SASL negotiation (Section 6), and
   servers SHOULD use TLS between two domains for the purpose of
   securing server-to-server communications.

The following rules apply:

1. An initiating entity that complies with this specification MUST
    include the 'version' attribute set to a value of "1.0" in the
    initial stream header.

2. If the TLS negotiation occurs between two servers, communications
    MUST NOT proceed until the Domain Name System (DNS) hostnames
    asserted by the servers have been resolved (see Server-to-Server
    Communications (Section 14.4)).

3. When a receiving entity that complies with this specification
    receives an initial stream header that includes the 'version'
    attribute set to a value of at least "1.0", after sending a
    stream header in reply (including the version flag), it MUST
    include a <starttls/> element (qualified by the
    'urn:ietf:params:xml:ns:xmpp-tls' namespace) along with the list
    of other stream features it supports.

4. If the initiating entity chooses to use TLS, TLS negotiation MUST
    be completed before proceeding to SASL negotiation; this order of
    negotiation is required to help safeguard authentication
    information sent during SASL negotiation, as well as to make it
    possible to base the use of the SASL EXTERNAL mechanism on a
    certificate provided during prior TLS negotiation.

5. During TLS negotiation, an entity MUST NOT send any white space
    characters (matching production [3] content of [XML]) within the
    root stream element as separators between elements (any white
    space characters shown in the TLS examples below are included for
    the sake of readability only); this prohibition helps to ensure
    proper security layer byte precision.

6. The receiving entity MUST consider the TLS negotiation to have
    begun immediately after sending the closing ">" character of the
    element. The initiating entity MUST consider the TLS
    negotiation to have begun immediately after receiving the closing
    ">" character of the element from the receiving
    entity.

7. The initiating entity MUST validate the certificate presented by
    the receiving entity; see Certificate Validation (Section 14.2)
    regarding certificate validation procedures.

8. Certificates MUST be checked against the hostname as provided by
    the initiating entity (e.g., a user), not the hostname as
    resolved via the Domain Name System; e.g., if the user specifies
    a hostname of "example.com" but a DNS SRV [SRV] lookup returned

"im.example.com", the certificate MUST be checked as
"example.com". If a JID for any kind of XMPP entity (e.g.,
client or server) is represented in a certificate, it MUST be
represented as a UTF8String within an otherName entity inside the
subjectAltName, using the [ASN.1] Object Identifier
"id-on-xmppAddr" specified in Section 5.1.1 of this document.

9. If the TLS negotiation is successful, the receiving entity MUST
    discard any knowledge obtained in an insecure manner from the
    initiating entity before TLS takes effect.

10. If the TLS negotiation is successful, the initiating entity MUST
    discard any knowledge obtained in an insecure manner from the
    receiving entity before TLS takes effect.

11. 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.

12. If the TLS negotiation is successful, the initiating entity MUST
    continue with SASL negotiation.

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

14. See Mandatory-to-Implement Technologies (Section 14.7) regarding
    mechanisms that MUST be supported.

5.1.1. ASN.1 Object Identifier for XMPP Address

The [ASN.1] Object Identifier "id-on-xmppAddr" described above is
defined as follows:

id-pkix OBJECT IDENTIFIER ::= { iso(1) identified-organization(3)
        dod(6) internet(1) security(5) mechanisms(5) pkix(7) }

id-on OBJECT IDENTIFIER ::= { id-pkix 8 } -- other name forms

id-on-xmppAddr OBJECT IDENTIFIER ::= { id-on 5 }

XmppAddr ::= UTF8String

This Object Identifier MAY also be represented in the dotted display
format as "1.3.6.1.5.5.7.8.5".

5.2. Narrative

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

   1. The initiating entity opens a TCP connection and initiates the
       stream by sending the opening XML stream header to the receiving
       entity, including the 'version' attribute set to a value of at
       least "1.0".

   2. The receiving entity responds by opening a TCP connection and
       sending an XML stream header to the initiating entity, including
       the 'version' attribute set to a value of at least "1.0".

   3. The receiving entity offers the STARTTLS extension to the
       initiating entity by including it with the list of other
       supported stream features (if TLS is required for interaction
       with the receiving entity, it SHOULD signal that fact by
       including a <required/> element as a child of the
       element).

   4. The initiating entity issues the STARTTLS command (i.e., a
       element qualified by the
       'urn:ietf:params:xml:ns:xmpp-tls' namespace) to instruct the
       receiving entity that it wishes to begin a TLS negotiation to
       secure the stream.

   5. The receiving entity MUST reply with either a element
       or a element qualified by the
       'urn:ietf:params:xml:ns:xmpp-tls' namespace. If the failure case
       occurs, the receiving entity MUST terminate both the XML stream
       and the underlying TCP connection. If the proceed case occurs,
       the entities MUST attempt to complete the TLS negotiation over
       the TCP connection and MUST NOT send any further XML data until
       the TLS negotiation is complete.

   6. The initiating entity and receiving entity attempt to complete a
       TLS negotiation in accordance with [TLS].

   7. If the TLS negotiation is unsuccessful, the receiving entity MUST
       terminate the TCP connection. If the TLS negotiation is
       successful, the initiating entity MUST initiate a new stream by
       sending an opening XML stream header to the receiving entity (it
       is not necessary to send a closing </stream> tag first, since the
       receiving entity and initiating entity MUST consider the original
       stream to be closed upon successful TLS negotiation).

   8. Upon receiving the new stream header from the initiating entity,
       the receiving entity MUST respond by sending a new XML stream
       header to the initiating entity along with the available features
       (but not including the STARTTLS feature).

5.3. Client-to-Server Example

   The following example shows the data flow for a client securing a
   stream using STARTTLS (note: the alternate steps shown below are
   provided to illustrate the protocol for failure cases; they are not
   exhaustive and would not necessarily be triggered by the data sent in
   the example).

   Step 1: Client initiates stream to server:

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

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

   <stream:stream
       xmlns='jabber:client'
       xmlns:stream='http://etherx.jabber.org/streams'
       id='c2s_123'
       from='example.com'
       version='1.0'>

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

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

   Step 4: Client sends the STARTTLS command to server:

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

Step 5: Server informs client that it is allowed to proceed:

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

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

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

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

Step 7: If TLS negotiation is successful, client initiates a new
stream to server:

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

Step 7 (alt): If TLS negotiation is unsuccessful, server closes TCP
connection.

Step 8: Server responds by sending a stream header to client along
with any available stream features:

```
<stream:stream
    xmlns='jabber:client'
    xmlns:stream='http://etherx.jabber.org/streams'
    from='example.com'
    id='c2s_234'
    version='1.0'>
<stream:features>
  <mechanisms xmlns='urn:ietf:params:xml:ns:xmpp-sasl'>
    <mechanism>DIGEST-MD5</mechanism>
    <mechanism>PLAIN</mechanism>
    <mechanism>EXTERNAL</mechanism>
  </mechanisms>
</stream:features>
```

Step 9: Client continues with SASL negotiation (Section 6).

5.4. Server-to-Server Example

   The following example shows the data flow for two servers securing a
   stream using STARTTLS (note: the alternate steps shown below are
   provided to illustrate the protocol for failure cases; they are not
   exhaustive and would not necessarily be triggered by the data sent in
   the example).

   Step 1: Server1 initiates stream to Server2:

   <stream:stream
       xmlns='jabber:server'
       xmlns:stream='http://etherx.jabber.org/streams'
       to='example.com'
       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'
       from='example.com'
       id='s2s_123'
       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='urn:ietf:params:xml:ns:xmpp-tls'>
     </starttls>
     <mechanisms xmlns='urn:ietf:params:xml:ns:xmpp-sasl'>
       <mechanism>DIGEST-MD5</mechanism>
       <mechanism>KERBEROS_V4</mechanism>
     </mechanisms>
   </stream:features>

   Step 4: Server1 sends the STARTTLS command to Server2:

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

   Step 5: Server2 informs Server1 that it is allowed to proceed:

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

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

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

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

   Step 7: If TLS negotiation is successful, Server1 initiates a new
   stream to Server2:

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

   Step 7 (alt): If TLS negotiation is unsuccessful, Server2 closes TCP
   connection.

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

   <stream:stream
       xmlns='jabber:server'
       xmlns:stream='http://etherx.jabber.org/streams'
       from='example.com'
       id='s2s_234'
       version='1.0'>
   <stream:features>
     <mechanisms xmlns='urn:ietf:params:xml:ns:xmpp-sasl'>
       <mechanism>DIGEST-MD5</mechanism>
       <mechanism>KERBEROS_V4</mechanism>
       <mechanism>EXTERNAL</mechanism>
     </mechanisms>
   </stream:features>

   Step 9: Server1 continues with SASL negotiation (Section 6).

6. Use of SASL

6.1. Overview

   XMPP includes a method for authenticating a stream by means of an
   XMPP-specific profile of the Simple Authentication and Security Layer
   (SASL) protocol [SASL]. SASL provides a generalized method for
   adding authentication support to connection-based protocols, and XMPP
   uses a generic XML namespace profile for SASL that conforms to the
   profiling requirements of [SASL].

   The following rules apply:

   1. If the SASL negotiation occurs between two servers,
       communications MUST NOT proceed until the Domain Name System
       (DNS) hostnames asserted by the servers have been resolved (see
       Server-to-Server Communications (Section 14.4)).

   2. If the initiating entity is capable of SASL negotiation, it MUST
       include the 'version' attribute set to a value of at least "1.0"
       in the initial stream header.

   3. If the receiving entity is capable of SASL negotiation, it MUST
       advertise one or more authentication mechanisms within a
       element qualified by the
       'urn:ietf:params:xml:ns:xmpp-sasl' namespace in reply to the
       opening stream tag received from the initiating entity (if the
       opening stream tag included the 'version' attribute set to a
       value of at least "1.0").

   4. During SASL negotiation, an entity MUST NOT send any white space
       characters (matching production [3] content of [XML]) within the
       root stream element as separators between elements (any white
       space characters shown in the SASL examples below are included
       for the sake of readability only); this prohibition helps to
       ensure proper security layer byte precision.

   5. Any XML character data contained within the XML elements used
       during SASL negotiation MUST be encoded using base64, where the
       encoding adheres to the definition in Section 3 of RFC 3548
       [BASE64].

   6. If provision of a "simple username" is supported by the selected
       SASL mechanism (e.g., this is supported by the DIGEST-MD5 and
       CRAM-MD5 mechanisms but not by the EXTERNAL and GSSAPI
       mechanisms), during authentication the initiating entity SHOULD
       provide as the simple username its sending domain (IP address or
       fully qualified domain name as contained in a domain identifier)

in the case of server-to-server communications or its registered
account name (user or node name as contained in an XMPP node
identifier) in the case of client-to-server communications.

7. If the initiating entity wishes to act on behalf of another
    entity and the selected SASL mechanism supports transmission of
    an authorization identity, the initiating entity MUST provide an
    authorization identity during SASL negotiation. If the
    initiating entity does not wish to act on behalf of another
    entity, it MUST NOT provide an authorization identity. As
    specified in [SASL], the initiating entity MUST NOT provide an
    authorization identity unless the authorization identity is
    different from the default authorization identity derived from
    the authentication identity as described in [SASL]. If provided,
    the value of the authorization identity MUST be of the form
    <domain> (i.e., a domain identifier only) for servers and of the
    form <node@domain> (i.e., node identifier and domain identifier)
    for clients.

8. Upon successful SASL negotiation that 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.

9. Upon successful SASL negotiation that 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.

10. See Mandatory-to-Implement Technologies (Section 14.7) regarding
    mechanisms that MUST be supported.

6.2. Narrative

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

   1. The initiating entity requests SASL authentication by including
       the '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 reply, the receiving entity
       advertises a list of available SASL authentication mechanisms;
       each of these is a <mechanism/> element included as a child
       within a <mechanisms/> container element qualified by the
       'urn:ietf:params:xml:ns:xmpp-sasl' namespace, which in turn is a
       child of a <features/> element in the streams namespace. If Use
       of TLS (Section 5) needs to 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 mechanisms prior to TLS negotiation. If the
initiating entity presents a valid certificate during prior TLS
negotiation, the receiving entity SHOULD offer the SASL EXTERNAL
mechanism to the initiating entity during SASL negotiation (refer
to [SASL]), although the EXTERNAL mechanism MAY be offered under
other circumstances as well.

3. The initiating entity selects a mechanism by sending an
    element qualified by the 'urn:ietf:params:xml:ns:xmpp-sasl'
    namespace to the receiving entity and including an appropriate
    value for the 'mechanism' attribute. This element MAY contain
    XML character data (in SASL terminology, the "initial response")
    if the mechanism supports or requires it; if the initiating
    entity needs to send a zero-length initial response, it MUST
    transmit the response as a single equals sign ("="), which
    indicates that the response is present but contains no data.

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

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

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

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

1. The initiating entity aborts the handshake by sending an
    element qualified by the 'urn:ietf:params:xml:ns:xmpp-sasl'
    namespace to the receiving entity. Upon receiving an
    element, the receiving entity SHOULD allow a configurable but
    reasonable number of retries (at least 2), after which it MUST
    terminate the TCP connection; this enables the initiating entity
    (e.g., an end-user client) to tolerate incorrectly-provided
    credentials (e.g., a mistyped password) without being forced to
    reconnect.

   2. The receiving entity reports failure of the handshake by sending
       a <failure/> element qualified by the
       'urn:ietf:params:xml:ns:xmpp-sasl' namespace to the initiating
       entity (the particular cause of failure SHOULD be communicated in
       an appropriate child element of the <failure/> element as defined
       under SASL Errors (Section 6.4)). If the failure case occurs,
       the receiving entity SHOULD allow a configurable but reasonable
       number of retries (at least 2), after which it MUST terminate the
       TCP connection; this enables the initiating entity (e.g., an
       end-user client) to tolerate incorrectly-provided credentials
       (e.g., a mistyped password) without being forced to reconnect.

   3. The receiving entity reports success of the handshake by sending
       a <success/> element qualified by the
       'urn:ietf:params:xml:ns:xmpp-sasl' namespace to the initiating
       entity; this element MAY contain XML character data (in SASL
       terminology, "additional data with success") if required by the
       chosen SASL mechanism. Upon receiving the <success/> element,
       the initiating entity MUST initiate a new stream by sending an
       opening XML stream header to the receiving entity (it is not
       necessary to send a closing </stream> tag first, since the
       receiving entity and initiating entity MUST consider the original
       stream to be closed upon sending or receiving the
       element). Upon receiving the new stream header from the
       initiating entity, the receiving entity MUST respond by sending a
       new XML stream header to the initiating entity, along with any
       available features (but not including the STARTTLS and SASL
       features) or an empty element (to signify that no
       additional features are available); any such additional features
       not defined herein MUST be defined by the relevant extension to
       XMPP.

6.3. SASL Definition

   The profiling requirements of [SASL] require that the following
   information be supplied by a protocol definition:

   service name: "xmpp"

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

exchange sequence: Challenges and responses are carried through the
exchange of <challenge/> elements from receiving entity to
initiating entity and <response/> elements from initiating entity
to receiving entity. The receiving entity reports failure by
sending a <failure/> element and success by sending a
element; the initiating entity aborts the exchange by sending an
element. Upon successful negotiation, both sides
consider the original XML stream to be closed and new stream
headers are sent by both entities.

security layer negotiation: The security layer takes effect
immediately after sending the closing ">" character of the
element for the receiving entity, and immediately after
receiving the closing ">" character of the element for
the initiating entity. The order of layers is first [TCP], then
[TLS], then [SASL], then XMPP.

use of the authorization identity: The authorization identity may be
used by xmpp to denote the non-default <node@domain> of a client
or the sending <domain> of a server.

6.4. SASL Errors

   The following SASL-related error conditions are defined:

   o -- The receiving entity acknowledges an
      element sent by the initiating entity; sent in reply to the
      element.

   o -- The data provided by the initiating
      entity could not be processed because the [BASE64] encoding is
      incorrect (e.g., because the encoding does not adhere to the
      definition in Section 3 of [BASE64]); sent in reply to a
      element or an element with initial response
      data.

   o -- The authzid provided by the initiating
      entity is invalid, either because it is incorrectly formatted or
      because the initiating entity does not have permissions to
      authorize that ID; sent in reply to a element or an
      element with initial response data.

   o -- The initiating entity did not provide a
      mechanism or requested a mechanism that is not supported by the
      receiving entity; sent in reply to an element.

   o <mechanism-too-weak/> -- The mechanism requested by the initiating
      entity is weaker than server policy permits for that initiating
      entity; sent in reply to a <response/> element or an <auth/>
      element with initial response data.

   o <not-authorized/> -- The authentication failed because the
      initiating entity did not provide valid credentials (this includes
      but is not limited to the case of an unknown username); sent in
      reply to a <response/> element or an <auth/> element with initial
      response data.

   o <temporary-auth-failure/> -- The authentication failed because of
      a temporary error condition within the receiving entity; sent in
      reply to an <auth/> element or <response/> element.

6.5. Client-to-Server Example

The following example shows the data flow for a client authenticating
with a server using SASL, normally after successful TLS negotiation
(note: the alternate steps shown below are provided to illustrate the
protocol for failure cases; they are not exhaustive and would not
necessarily be triggered by the data sent in the example).

Step 1: Client initiates stream to server:

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

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

```
<stream:stream
    xmlns='jabber:client'
    xmlns:stream='http://etherx.jabber.org/streams'
    id='c2s_234'
    from='example.com'
    version='1.0'>
```

Step 3: Server informs client of available authentication mechanisms:

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

    Step 4: Client selects an authentication mechanism:

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

    Step 5: Server sends a [BASE64] encoded challenge to client:

    <challenge xmlns='urn:ietf:params:xml:ns:xmpp-sasl'>
    cmVhbG09InNvbWVyZWFsbSIsbm9uY2U9Ik9BNk1HOXRFUUdtMmhoIixxb3A9ImF1dGgi
    LGNoYXJzZXQ9dXRmLTgsYWxnb3JpdGhtPW1kNS1zZXNzCg==
    </challenge>

    The decoded challenge is:

    realm="somerealm",nonce="OA6MG9tEQGm2hh",\
    qop="auth",charset=utf-8,algorithm=md5-sess

    Step 5 (alt): Server returns error to client:

    <failure xmlns='urn:ietf:params:xml:ns:xmpp-sasl'>
      <incorrect-encoding/>
    </failure>
    </stream:stream>

    Step 6: Client sends a [BASE64] encoded response to the challenge:

    <response xmlns='urn:ietf:params:xml:ns:xmpp-sasl'>
    dXNlcm5hbWU9InNvbWVub2RlIixyZWFsbT0ic29tZXJlYWxtIixub25jZT0i
    T0E2TUc5dEVRR20yaGgiLGNub25jZT0iT0E2TUhYaDZWcVRyUmsiLG5jPTAw
    MDAwMDAxLHFvcD1hdXRoLGRpZ2VzdC11cmk9InhtcHAvZXhhbXBsZS5jb20i
    LHJlc3BvbnNlPWQzODhkYWQ5MGQ0YmJkNzYwYTE1MjMyMWYyMTQzYWY3LGNo
    YXJzZXQ9dXRmLTgK
    </response>

    The decoded response is:

    username="somenode",realm="somerealm",\
    nonce="OA6MG9tEQGm2hh",cnonce="OA6MHXh6VqTrRk",\
    nc=00000001,qop=auth,digest-uri="xmpp/example.com",\
    response=d388dad90d4bbd760a152321f2143af7,charset=utf-8

    Step 7: Server sends another [BASE64] encoded challenge to client:

    <challenge xmlns='urn:ietf:params:xml:ns:xmpp-sasl'>
    cnNwYXV0aD1lYTQwZjYwMzM1YzQyN2I1NTI3Yjg0ZGJhYmNkZmZmZAo=
    </challenge>

   The decoded challenge is:

   rspauth=ea40f60335c427b5527b84dbabcdfffd

   Step 7 (alt): Server returns error to client:

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

   Step 8: Client responds to the challenge:

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

   Step 9: Server informs client of successful authentication:

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

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

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

   Step 10: Client initiates a new stream to server:

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

   Step 11: Server responds by sending a stream header to client along
   with any additional features (or an empty features element):

```
<stream:stream
    xmlns='jabber:client'
    xmlns:stream='http://etherx.jabber.org/streams'
    id='c2s_345'
    from='example.com'
    version='1.0'>
<stream:features>
  <bind xmlns='urn:ietf:params:xml:ns:xmpp-bind'/>
  <session xmlns='urn:ietf:params:xml:ns:xmpp-session'/>
</stream:features>
```

6.6. Server-to-Server Example

   The following example shows the data flow for a server authenticating
   with another server using SASL, normally after successful TLS
   negotiation (note: the alternate steps shown below are provided to
   illustrate the protocol for failure cases; they are not exhaustive
   and would not necessarily be triggered by the data sent in the
   example).

   Step 1: Server1 initiates stream to Server2:

   <stream:stream
       xmlns='jabber:server'
       xmlns:stream='http://etherx.jabber.org/streams'
       to='example.com'
       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'
       from='example.com'
       id='s2s_234'
       version='1.0'>

   Step 3: Server2 informs Server1 of available authentication
   mechanisms:

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

   Step 4: Server1 selects an authentication mechanism:

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

   Step 5: Server2 sends a [BASE64] encoded challenge to Server1:

   <challenge xmlns='urn:ietf:params:xml:ns:xmpp-sasl'>
   cmVhbG09InNvbWVyZWFsbSIsbm9uY2U9Ik9BNk1HOXRFUUdtMmhoIixxb3A9
   ImF1dGgiLGNoYXJzZXQ9dXRmLTgsYWxnb3JpdGhtPW1kNS1zZXNz
   </challenge>

   The decoded challenge is:

   realm="somerealm",nonce="OA6MG9tEQGm2hh",\
   qop="auth",charset=utf-8,algorithm=md5-sess

   Step 5 (alt): Server2 returns error to Server1:

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

   Step 6: Server1 sends a [BASE64] encoded response to the challenge:

   <response xmlns='urn:ietf:params:xml:ns:xmpp-sasl'>
   dXNlcm5hbWU9ImV4YW1wbGUub3JnIixyZWFsbT0ic29tZXJlYWxtIixub25j
   ZT0iT0E2TUc5dEVRR20yaGgiLGNub25jZT0iT0E2TUhYaDZWcVRyUmsiLG5j
   PTAwMDAwMDAxLHFvcD1hdXRoLGRpZ2VzdC11cmk9InhtcHAvZXhhbXBsZS5v
   cmciLHJlc3BvbnNlPWQzODhkYWQ5MGQ0YmJkNzYwYTE1MjMyMWYyMTQzYWY3
   LGNoYXJzZXQ9dXRmLTgK
   </response>

   The decoded response is:

   username="example.org",realm="somerealm",\
   nonce="OA6MG9tEQGm2hh",cnonce="OA6MHXh6VqTrRk",\
   nc=00000001,qop=auth,digest-uri="xmpp/example.org",\
   response=d388dad90d4bbd760a152321f2143af7,charset=utf-8

   Step 7: Server2 sends another [BASE64] encoded challenge to Server1:

   <challenge xmlns='urn:ietf:params:xml:ns:xmpp-sasl'>
   cnNwYXV0aD1lYTQwZjYwMzM1YzQyN2I1NTI3Yjg0ZGJhYmNkZmZmZAo=
   </challenge>

   The decoded challenge is:

   rspauth=ea40f60335c427b5527b84dbabcdfffd

   Step 7 (alt): Server2 returns error to Server1:

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

Step 8: Server1 responds to the challenge:

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

Step 8 (alt): Server1 aborts negotiation:

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

Step 9: Server2 informs Server1 of successful authentication:

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

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

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

Step 10: Server1 initiates a new stream to Server2:

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

Step 11: Server2 responds by sending a stream header to Server1 along
with any additional features (or an empty features element):

```
<stream:stream
    xmlns='jabber:client'
    xmlns:stream='http://etherx.jabber.org/streams'
    from='example.com'
    id='s2s_345'
    version='1.0'>
<stream:features/>
```

7. Resource Binding

   After SASL negotiation (Section 6) with the receiving entity, the
   initiating entity MAY want or need to bind a specific resource to
   that stream. In general this applies only to clients: in order to
   conform to the addressing format (Section 3) and stanza delivery
   rules (Section 10) specified herein, there MUST be a resource
   identifier associated with the <node@domain> of the client (which is

either generated by the server or provided by the client
application); this ensures that the address for use over that stream
is a "full JID" of the form <node@domain/resource>.

Upon receiving a success indication within the SASL negotiation, the
client MUST send a new stream header to the server, to which the
server MUST respond with a stream header as well as a list of
available stream features. Specifically, if the server requires the
client to bind a resource to the stream after successful SASL
negotiation, it MUST include an empty <bind/> element qualified by
the 'urn:ietf:params:xml:ns:xmpp-bind' namespace in the stream
features list it presents to the client upon sending the header for
the response stream sent after successful SASL negotiation (but not
before):

Server advertises resource binding feature to client:

```
<stream:stream
    xmlns='jabber:client'
    xmlns:stream='http://etherx.jabber.org/streams'
    id='c2s_345'
    from='example.com'
    version='1.0'>
<stream:features>
  <bind xmlns='urn:ietf:params:xml:ns:xmpp-bind'/>
</stream:features>
```

Upon being so informed that resource binding is required, the client
MUST bind a resource to the stream by sending to the server an IQ
stanza of type "set" (see IQ Semantics (Section 9.2.3)) containing
data qualified by the 'urn:ietf:params:xml:ns:xmpp-bind' namespace.

If the client wishes to allow the server to generate the resource
identifier on its behalf, it sends an IQ stanza of type "set" that
contains an empty <bind/> element:

Client asks server to bind a resource:

```
<iq type='set' id='bind_1'>
  <bind xmlns='urn:ietf:params:xml:ns:xmpp-bind'/>
</iq>
```

A server that supports resource binding MUST be able to generate a
resource identifier on behalf of a client. A resource identifier
generated by the server MUST be unique for that <node@domain>.

If the client wishes to specify the resource identifier, it sends an
IQ stanza of type "set" that contains the desired resource identifier
as the XML character data of a <resource/> element that is a child of
the <bind/> element:

Client binds a resource:

```
<iq type='set' id='bind_2'>
  <bind xmlns='urn:ietf:params:xml:ns:xmpp-bind'>
    <resource>someresource</resource>
  </bind>
</iq>
```

Once the server has generated a resource identifier for the client or
accepted the resource identifier provided by the client, it MUST
return an IQ stanza of type "result" to the client, which MUST
include a <jid/> child element that specifies the full JID for the
connected resource as determined by the server:

Server informs client of successful resource binding:

```
<iq type='result' id='bind_2'>
  <bind xmlns='urn:ietf:params:xml:ns:xmpp-bind'>
    <jid>somenode@example.com/someresource</jid>
  </bind>
</iq>
```

A server SHOULD accept the resource identifier provided by the
client, but MAY override it with a resource identifier that the
server generates; in this case, the server SHOULD NOT return a stanza
error (e.g., <forbidden/>) to the client but instead SHOULD
communicate the generated resource identifier to the client in the IQ
result as shown above.

When a client supplies a resource identifier, the following stanza
error conditions are possible (see Stanza Errors (Section 9.3)):

o The provided resource identifier cannot be processed by the server
   in accordance with Resourceprep (Appendix B).

o The client is not allowed to bind a resource to the stream (e.g.,
   because the node or user has reached a limit on the number of
   connected resources allowed).

o The provided resource identifier is already in use but the server
   does not allow binding of multiple connected resources with the
   same identifier.

The protocol for these error conditions is shown below.

Resource identifier cannot be processed:

```
<iq type='error' id='bind_2'>
  <bind xmlns='urn:ietf:params:xml:ns:xmpp-bind'>
    <resource>someresource</resource>
  </bind>
  <error type='modify'>
    <bad-request xmlns='urn:ietf:params:xml:ns:xmpp-stanzas'/>
  </error>
</iq>
```

Client is not allowed to bind a resource:

```
<iq type='error' id='bind_2'>
  <bind xmlns='urn:ietf:params:xml:ns:xmpp-bind'>
    <resource>someresource</resource>
  </bind>
  <error type='cancel'>
    <not-allowed xmlns='urn:ietf:params:xml:ns:xmpp-stanzas'/>
  </error>
</iq>
```

Resource identifier is in use:

```
<iq type='error' id='bind_2'>
  <bind xmlns='urn:ietf:params:xml:ns:xmpp-bind'>
    <resource>someresource</resource>
  </bind>
  <error type='cancel'>
    <conflict xmlns='urn:ietf:params:xml:ns:xmpp-stanzas'/>
  </error>
</iq>
```

If, before completing the resource binding step, the client attempts
to send an XML stanza other than an IQ stanza with a <bind/> child
qualified by the 'urn:ietf:params:xml:ns:xmpp-bind' namespace, the
server MUST NOT process the stanza and SHOULD return a
stanza error to the client.

8. Server Dialback

8.1. Overview

   The Jabber protocols from which XMPP was adapted include a "server
   dialback" method for protecting against domain spoofing, thus making
   it more difficult to spoof XML stanzas. Server dialback is not a
   security mechanism, and results in weak verification of server
   identities only (see Server-to-Server Communications (Section 14.4)
   regarding this method's security characteristics). Domains requiring
   robust security SHOULD use TLS and SASL; see Server-to-Server
   Communications (Section 14.4) for details. If SASL is used for
   server-to-server authentication, dialback SHOULD NOT be used since it
   is unnecessary. Documentation of dialback is included mainly for the
   sake of backward-compatibility with existing implementations and
   deployments.

   The server dialback method is made possible by the existence of the
   Domain Name System (DNS), since one server can (normally) discover
   the authoritative server for a given domain. Because dialback
   depends on DNS, inter-domain communications MUST NOT proceed until
   the Domain Name System (DNS) hostnames asserted by the servers have
   been resolved (see Server-to-Server Communications (Section 14.4)).

   Server dialback is uni-directional, and results in (weak)
   verification of identities for one stream in one direction. Because
   server dialback is not an authentication mechanism, mutual
   authentication is not possible via dialback. Therefore, server
   dialback MUST be completed in each direction in order to enable
   bi-directional communications between two domains.

   The method for generating and verifying the keys used in server
   dialback MUST take into account the hostnames being used, the stream
   ID generated by the receiving server, and a secret known by the
   authoritative server's network. The stream ID is security-critical
   in server dialback and therefore MUST be both unpredictable and
   non-repeating (see [RANDOM] for recommendations regarding randomness
   for security purposes).

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

   The following terminology applies:

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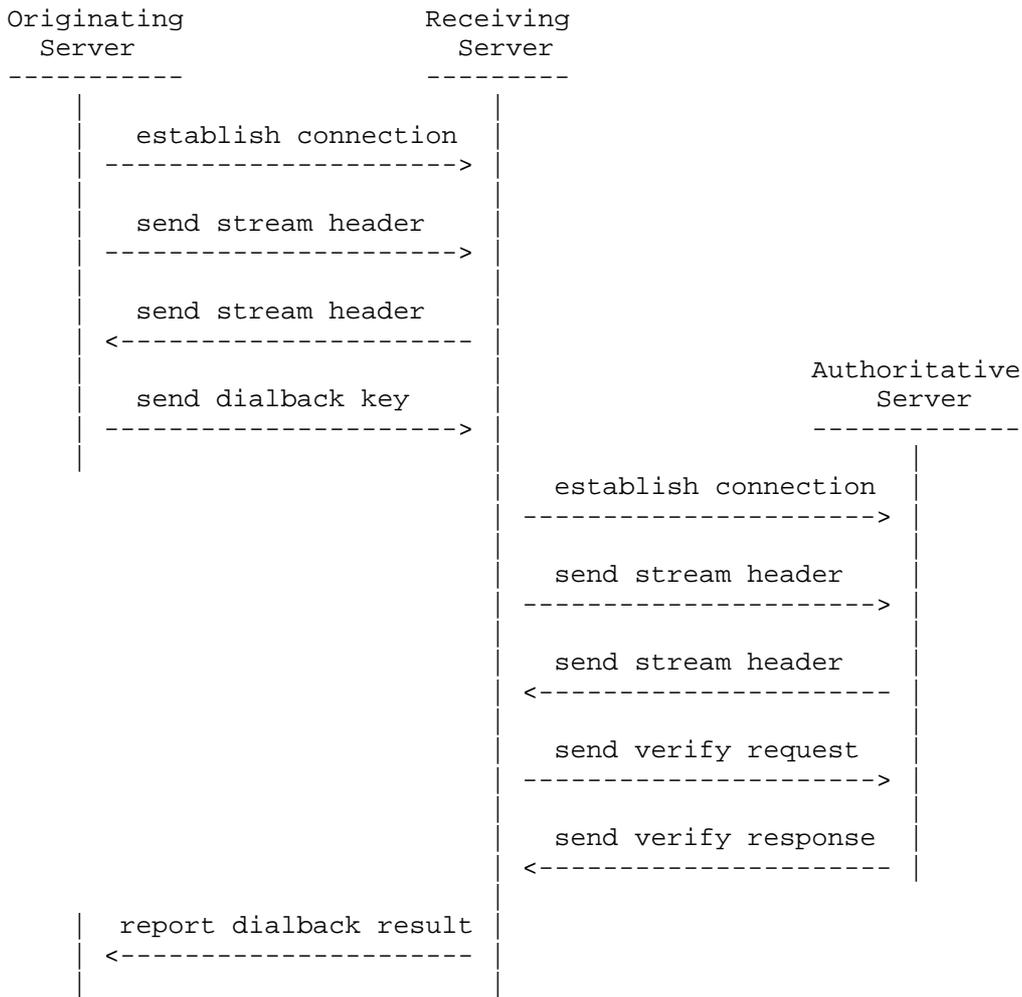   connection between two domains.

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

   o Authoritative Server -- the server that answers to the DNS
      hostname asserted by the Originating Server; for basic
      environments this will be the Originating Server, but it could be
      a separate machine in the Originating Server's network.

8.2. Order of Events

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

   1. The Originating Server establishes a connection to the Receiving
       Server.

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

   3. The Receiving Server establishes a connection to the
       Authoritative Server.

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

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

   6. The Receiving Server informs the 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      |
        | --------------------------> |
        |                             |
        |     send stream header      |                    Authoritative
        | <-------------------------- |                       Server
        |                             |                    -------------
        |     send dialback key       |
        | --------------------------> |                    -------------
        |                             |                          |
        |                             |   establish connection   |
        |                             | -----------------------> |
        |                             |                          |
        |                             |     send stream header    |
        |                             | -----------------------> |
        |                             |                          |
        |                             |     send stream header    |
        |                             | <----------------------- |
        |                             |                          |
        |                             |     send verify request   |
        |                             | -----------------------> |
        |                             |                          |
        |                             |     send verify response  |
        |                             | <----------------------- |
        |                             |                          |
        |   report dialback result    |
        | <-------------------------- |
        |                             |
```

8.3. Protocol

   The detailed protocol interaction between the servers is as follows:

   1. The Originating Server establishes TCP connection to the
       Receiving Server.

   2. The Originating Server sends a stream header to the Receiving
       Server:

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

   Note: The 'to' and 'from' attributes are OPTIONAL on the root stream
   element. The inclusion of the xmlns:db namespace declaration with
   the name shown indicates to the Receiving Server that the Originating
   Server supports dialback. If the namespace name is incorrect, then
   the Receiving Server MUST generate an <invalid-namespace/> stream
   error condition and terminate both the XML stream and the underlying
   TCP connection.

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

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

   Note: The 'to' and 'from' attributes are OPTIONAL on the root stream
   element. If the namespace name is incorrect, then the Originating
   Server MUST generate an <invalid-namespace/> stream error condition
   and terminate both the XML stream and the underlying TCP connection.
   Note well that the Receiving Server SHOULD reply but MAY silently
   terminate the XML stream and underlying TCP connection depending on
   security policies in place; however, if the Receiving Server desires
   to proceed, it MUST send a stream header back to the Originating
   Server.

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

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

   Note: This key is not examined by the Receiving Server, since the
   Receiving Server does not keep information about the Originating
   Server between sessions. The key generated by the Originating Server
   MUST be based in part on the value of the ID provided by the

Receiving Server in the previous step, and in part on a secret shared
by the Originating Server and Authoritative Server. If the value of
the 'to' address does not match a hostname recognized by the
Receiving Server, then the Receiving Server MUST generate a
stream error condition and terminate both the XML
stream and the underlying TCP connection. If the value of the 'from'
address matches a domain with which the Receiving Server already has
an established connection, then the Receiving Server MUST maintain
the existing connection until it validates whether the new connection
is legitimate; additionally, the Receiving Server MAY choose to
generate a <not-authorized/> stream error condition for the new
connection and then terminate both the XML stream and the underlying
TCP connection related to the new request.

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

6. The Receiving Server sends the Authoritative Server a stream
    header:

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

Note: The 'to' and 'from' attributes are OPTIONAL on the root stream
element. If the namespace name is incorrect, then the Authoritative
Server MUST generate an <invalid-namespace/> stream error condition
and terminate both the XML stream and the underlying TCP connection.

7. The Authoritative Server sends the Receiving Server a stream
    header:

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

Note: If the namespace name is incorrect, then the Receiving Server
MUST generate an <invalid-namespace/> stream error condition and
terminate both the XML stream and the underlying TCP connection
between it and the Authoritative Server. If a stream error occurs
between the Receiving Server and the Authoritative Server, then the
Receiving Server MUST generate a <remote-connection-failed/> stream

error condition and terminate both the XML stream and the underlying
TCP connection between it and the Originating Server.

8. The Receiving Server sends the Authoritative Server a request for
    verification of 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 the
Receiving Server's stream header to the Originating Server in Step 3,
and the key that the Originating Server sent to the Receiving Server
in Step 4. Based on this information, as well as shared secret
information within the Authoritative Server's network, the key is
verified. Any verifiable method MAY be used to generate the key. If
the value of the 'to' address does not match a hostname recognized by
the Authoritative Server, then the Authoritative Server MUST generate
a <host-unknown/> stream error condition and terminate both the XML
stream and the underlying TCP connection. If the value of the 'from'
address does not match the hostname represented by the Receiving
Server when opening the TCP connection (or any validated domain
thereof, such as a validated subdomain of the Receiving Server's
hostname or another validated domain hosted by the Receiving Server),
then the Authoritative Server MUST generate an <invalid-from/> stream
error condition and terminate both the XML stream and the underlying
TCP connection.

9. The Authoritative Server verifies whether the key was valid or
    invalid:

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

or

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

Note: If the ID does not match that provided by the Receiving Server
in Step 3, then the Receiving Server MUST generate an
stream error condition and terminate both the XML stream and the
underlying TCP connection. If the value of the 'to' address does not
match a hostname recognized by the Receiving Server, then the
Receiving Server MUST generate a stream error
condition and terminate both the XML stream and the underlying TCP
connection. If the value of the 'from' address does not match the
hostname represented by the Originating Server when opening the TCP
connection (or any validated domain thereof, such as a validated
subdomain of the Originating Server's hostname or another validated
domain hosted by the Originating Server), then the Receiving Server
MUST generate an <invalid-from/> stream error condition and terminate
both the XML stream and the underlying TCP connection. After
returning the verification to the Receiving Server, the Authoritative
Server SHOULD terminate the stream between them.

10. The Receiving Server informs the Originating Server of the
    result:

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

Note: At this point, the connection has either been validated via a
type='valid', or reported as invalid. If the connection is invalid,
then the Receiving Server MUST terminate both the XML stream and the
underlying TCP connection. If the connection is validated, data can
be sent by the Originating Server and read by the Receiving Server;
before that, all XML stanzas sent to the Receiving Server SHOULD be
silently dropped.

The result of the foregoing is that the Receiving Server has verified
the identity of the Originating Server, so that the Originating
Server can send, and the Receiving Server can accept, XML stanzas
over the "initial stream" (i.e., the stream from the Originating
Server to the Receiving Server). In order to verify the identities
of the entities using the "response stream" (i.e., the stream from
the Receiving Server to the Originating Server), dialback MUST be
completed in the opposite direction as well.

After successful dialback negotiation, the Receiving Server SHOULD
accept subsequent <db:result/> packets (e.g., validation requests
sent to a subdomain or other hostname serviced by the Receiving
Server) from the Originating Server over the existing validated
connection; this enables "piggybacking" of the original validated
connection in one direction.

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

9. XML Stanzas

   After TLS negotiation (Section 5) if desired, SASL negotiation
   (Section 6), and Resource Binding (Section 7) if necessary, XML
   stanzas can be sent over the streams. Three kinds of XML stanza are
   defined for the 'jabber:client' and 'jabber:server' namespaces:
   , , and . In addition, there are five
   common attributes for these kinds of stanza. These common
   attributes, as well as the basic semantics of the three stanza kinds,
   are defined herein; more detailed information regarding the syntax of
   XML stanzas in relation to instant messaging and presence
   applications is provided in [XMPP-IM].

9.1. Common Attributes

   The following five attributes are common to message, presence, and IQ
   stanzas:

9.1.1. 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) SHOULD NOT possess a 'to' attribute.

   In the 'jabber:server' namespace, a stanza MUST possess a 'to'
   attribute; if a server receives a stanza that does not meet this
   restriction, it MUST generate an <improper-addressing/> stream error
   condition and terminate both the XML stream and the underlying TCP
   connection with the offending server.

If the value of the 'to' attribute is invalid or cannot be contacted, the entity discovering that fact (usually the sender's or recipient's server) MUST return an appropriate error to the sender, setting the 'from' attribute of the error stanza to the value provided in the 'to' attribute of the offending stanza.

9.1.2. from

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

When a server receives an XML stanza within the context of an authenticated stream qualified by the 'jabber:client' namespace, it MUST do one of the following:

1. validate that the value of the 'from' attribute provided by the client is that of a connected resource for the associated entity

2. add a 'from' address to the stanza whose value is the bare JID (<node@domain>) or the full JID (<node@domain/resource>) determined by the server for the connected resource that generated the stanza (see Determination of Addresses (Section 3.5))

If a client attempts to send an XML stanza for which the value of the 'from' attribute does not match one of the connected resources for that entity, the server SHOULD return an <invalid-from/> stream error to the client. If a client attempts to send an XML stanza over a stream that is not yet authenticated, the server SHOULD return a <not-authorized/> stream error to the client. If generated, both of these conditions MUST result in closure of the stream and termination of the underlying TCP connection; this helps to prevent a denial of service attack launched from a rogue client.

When a server generates a stanza from the server itself for delivery to a connected client (e.g., in the context of data storage services provided by the server on behalf of the client), the stanza MUST either (1) not include a 'from' attribute or (2) include a 'from' attribute whose value is the account's bare JID (<node@domain>) or client's full JID (<node@domain/resource>). A server MUST NOT send to the client a stanza without a 'from' attribute if the stanza was not generated by the server itself. When a client receives a stanza that does not include a 'from' attribute, it MUST assume that the stanza is from the server to which the client is connected.

In the 'jabber:server' namespace, a stanza MUST possess a 'from' attribute; if a server receives a stanza that does not meet this restriction, it MUST generate an <improper-addressing/> stream error condition. Furthermore, the domain identifier portion of the JID

contained in the 'from' attribute MUST match the hostname of the
sending server (or any validated domain thereof, such as a validated
subdomain of the sending server's hostname or another validated
domain hosted by the sending server) as communicated in the SASL
negotiation or dialback negotiation; if a server receives a stanza
that does not meet this restriction, it MUST generate an
stream error condition. Both of these conditions
MUST result in closing of the stream and termination of the
underlying TCP connection; this helps to prevent a denial of service
attack launched from a rogue server.

9.1.3. id

The optional 'id' attribute MAY be used by a sending entity for
internal tracking of stanzas that it sends and receives (especially
for tracking the request-response interaction inherent in the
semantics of IQ stanzas). It is OPTIONAL for the value of the 'id'
attribute to be unique globally, within a domain, or within a stream.
The semantics of IQ stanzas impose additional restrictions; see IQ
Semantics (Section 9.2.3).

9.1.4. 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; the values for message and
presence stanzas are specific to instant messaging and presence
applications and therefore are defined in [XMPP-IM], whereas the
values for IQ stanzas specify the role of an IQ stanza in a
structured request-response "conversation" and thus are defined under
IQ Semantics (Section 9.2.3) below. The only 'type' value common to
all three stanzas is "error"; see Stanza Errors (Section 9.3).

9.1.5. xml:lang

A stanza SHOULD possess an 'xml:lang' attribute (as defined in
Section 2.12 of [XML]) if the stanza contains XML character data that
is intended to be presented to a human user (as explained in RFC 2277
[CHARSET], "internationalization is for humans"). The value of the
'xml:lang' attribute specifies the default language of any such
human-readable XML character data, which MAY be overridden by the
'xml:lang' attribute of a specific child element. If a stanza does
not possess an 'xml:lang' attribute, an implementation MUST assume
that the default language is that specified for the stream as defined
under Stream Attributes (Section 4.4) above. The value of the
'xml:lang' attribute MUST be an NMTOKEN and MUST conform to the
format defined in RFC 3066 [LANGTAGS].

9.2. Basic Semantics
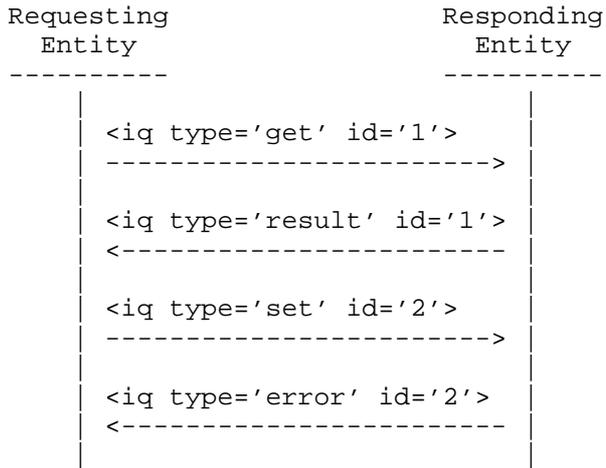
9.2.1. Message Semantics

   The <message/> stanza kind 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. All message
   stanzas SHOULD possess a 'to' attribute that specifies the intended
   recipient of the message; upon receiving such a stanza, a server
   SHOULD route or deliver it to the intended recipient (see Server
   Rules for Handling XML Stanzas (Section 10) for general routing and
   delivery rules related to XML stanzas).

9.2.2. Presence Semantics

   The <presence/> element can be seen as a basic broadcast or
   "publish-subscribe" mechanism, whereby multiple entities receive
   information about an entity to which they have subscribed (in this
   case, network availability information). In general, a publishing
   entity SHOULD send a presence stanza with no 'to' attribute, in which
   case the server to which the entity is connected SHOULD broadcast or
   multiplex that stanza to all subscribing entities. However, a
   publishing entity MAY also send a presence stanza with a 'to'
   attribute, in which case the server SHOULD route or deliver that
   stanza to the intended recipient. See Server Rules for Handling XML
   Stanzas (Section 10) for general routing and delivery rules related
   to XML stanzas, and [XMPP-IM] for presence-specific rules in the
   context of an instant messaging and presence application.

9.2.3. IQ Semantics

   Info/Query, or IQ, is a request-response mechanism, similar in some
   ways to [HTTP]. The semantics of IQ 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. Thus, IQ interactions follow a common pattern of
   structured data exchange such as get/result or set/result (although
   an error may be returned in reply 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='error' id='2'> |
        | <----------------------  |
        |                          |
```

In order to enforce these semantics, the following rules apply:

1. The 'id' attribute is REQUIRED for IQ stanzas.

2. The 'type' attribute is REQUIRED for IQ stanzas. The value MUST
    be one of the following:

    *   get -- The stanza is a request for information or
        requirements.

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

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

    *   error -- An error has occurred regarding processing or
        delivery of a previously-sent get or set (see Stanza Errors
        (Section 9.3)).

3. An entity that receives an IQ request of type "get" or "set" MUST
    reply with an IQ response of type "result" or "error" (the
    response MUST preserve the 'id' attribute of the request).

4. 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).

   5. An IQ stanza of type "get" or "set" MUST contain one and only one
       child element that specifies the semantics of the particular
       request or response.

   6. An IQ stanza of type "result" MUST include zero or one child
       elements.

   7. An IQ stanza of type "error" SHOULD include the child element
       contained in the associated "get" or "set" and MUST include an
       child; for details, see Stanza Errors (Section 9.3).

9.3. Stanza Errors

   Stanza-related errors are handled in a manner similar to stream
   errors (Section 4.7). However, unlike stream errors, stanza errors
   are recoverable; therefore error stanzas include hints regarding
   actions that the original sender can take in order to remedy the
   error.

9.3.1. Rules

   The following rules apply to stanza-related errors:

   o The receiving or processing entity that detects an error condition
      in relation to a stanza MUST return to the sending entity a stanza
      of the same kind (message, presence, or IQ), whose 'type'
      attribute is set to a value of "error" (such a stanza is called an
      "error stanza" herein).

   o The entity that generates an error stanza SHOULD include the
      original XML sent so that the sender can inspect and, if
      necessary, correct the XML before attempting to resend.

   o An error stanza MUST contain an <error/> child element.

   o An <error/> child MUST NOT be included if the 'type' attribute has
      a value other than "error" (or if there is no 'type' attribute).

   o An entity that receives an error stanza MUST NOT respond to the
      stanza with a further error stanza; this helps to prevent looping.

9.3.2. Syntax

   The syntax for stanza-related errors is as follows:

   <stanza-kind to='sender' type='error'>
     [RECOMMENDED to include sender XML here]
     <error type='error-type'>
       <defined-condition xmlns='urn:ietf:params:xml:ns:xmpp-stanzas'/>
       <text xmlns='urn:ietf:params:xml:ns:xmpp-stanzas'
             xml:lang='langcode'>
         OPTIONAL descriptive text
       </text>
       [OPTIONAL application-specific condition element]
     </error>
   </stanza-kind>

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

   The value of the <error/> element's 'type' attribute MUST be one of
   the following:

   o cancel -- do not retry (the error is unrecoverable)
   o continue -- proceed (the condition was only a warning)
   o modify -- retry after changing the data sent
   o auth -- retry after providing credentials
   o wait -- retry after waiting (the error is temporary)

   The <error/> element:

   o MUST contain a child element corresponding to one of the defined
      stanza error conditions specified below; this element MUST be
      qualified by the 'urn:ietf:params:xml:ns:xmpp-stanzas' namespace.

   o MAY contain a <text/> child containing XML character data that
      describes the error in more detail; this element MUST be qualified
      by the 'urn:ietf:params:xml:ns:xmpp-stanzas' namespace and SHOULD
      possess an 'xml:lang' attribute.

   o MAY contain a child element for an application-specific error
      condition; this element MUST be qualified by an
      application-defined namespace, and its structure is defined by
      that namespace.

   The <text/> element is OPTIONAL. If included, it SHOULD be used only
   to provide descriptive or diagnostic information that supplements the
   meaning of a defined condition or application-specific condition. It
   SHOULD NOT be interpreted programmatically by an application. It

SHOULD NOT be used as the error message presented to a user, but MAY
be shown in addition to the error message associated with the
included condition element (or elements).

Finally, to maintain backward compatibility, the schema (specified in
[XMPP-IM]) allows the optional inclusion of a 'code' attribute on the
element.

9.3.3. Defined Conditions

The following conditions are defined for use in stanza errors.

o -- the sender has sent XML that is malformed or
   that cannot be processed (e.g., an IQ stanza that includes an
   unrecognized value of the 'type' attribute); the associated error
   type SHOULD be "modify".

o -- access cannot be granted because an existing
   resource or session exists with the same name or address; the
   associated error type SHOULD be "cancel".

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

o -- the requesting entity does not possess the
   required permissions to perform the action; the associated error
   type SHOULD be "auth".

o -- the recipient or server can no longer be contacted at
   this address (the error stanza MAY contain a new address in the
   XML character data of the element); the associated error
   type SHOULD be "modify".

o -- the server could not process the
   stanza because of a misconfiguration or an otherwise-undefined
   internal server error; the associated error type SHOULD be "wait".

o -- the addressed JID or item requested cannot be
   found; the associated error type SHOULD be "cancel".

o -- the sending entity has provided or
   communicated an XMPP address (e.g., a value of the 'to' attribute)
   or aspect thereof (e.g., a resource identifier) that does not
   adhere to the syntax defined in Addressing Scheme (Section 3); the
   associated error type SHOULD be "modify".

   o <not-acceptable/> -- the recipient or server understands the
      request but is refusing to process it because it does not meet
      criteria defined by the recipient or server (e.g., a local policy
      regarding acceptable words in messages); the associated error type
      SHOULD be "modify".

   o <not-allowed/> -- the recipient or server does not allow any
      entity to perform the action; the associated error type SHOULD be
      "cancel".

   o <not-authorized/> -- the sender must provide proper credentials
      before being allowed to perform the action, or has provided
      improper credentials; the associated error type SHOULD be "auth".

   o <payment-required/> -- the requesting entity is not authorized to
      access the requested service because payment is required; the
      associated error type SHOULD be "auth".

   o <recipient-unavailable/> -- the intended recipient is temporarily
      unavailable; the associated error type SHOULD be "wait" (note: an
      application MUST NOT return this error if doing so would provide
      information about the intended recipient's network availability to
      an entity that is not authorized to know such information).

   o <redirect/> -- the recipient or server is redirecting requests for
      this information to another entity, usually temporarily (the error
      stanza SHOULD contain the alternate address, which MUST be a valid
      JID, in the XML character data of the <redirect/> element); the
      associated error type SHOULD be "modify".

   o <registration-required/> -- the requesting entity is not
      authorized to access the requested service because registration is
      required; the associated error type SHOULD be "auth".

   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 error type SHOULD be "cancel".

   o <remote-server-timeout/> -- a remote server or service specified
      as part or all of the JID of the intended recipient (or required
      to fulfill a request) could not be contacted within a reasonable
      amount of time; the associated error type SHOULD be "wait".

   o <resource-constraint/> -- the server or recipient lacks the system
      resources necessary to service the request; the associated error
      type SHOULD be "wait".

   o <service-unavailable/> -- the server or recipient does not
      currently provide the requested service; the associated error type
      SHOULD be "cancel".

   o <subscription-required/> -- the requesting entity is not
      authorized to access the requested service because a subscription
      is required; the associated error type SHOULD be "auth".

   o <undefined-condition/> -- the error condition is not one of those
      defined by the other conditions in this list; any error type may
      be associated with this condition, and it SHOULD be used only in
      conjunction with an application-specific condition.

   o <unexpected-request/> -- the recipient or server understood the
      request but was not expecting it at this time (e.g., the request
      was out of order); the associated error type SHOULD be "wait".

9.3.4. Application-Specific Conditions

   As noted, an application MAY provide application-specific stanza
   error information by including a properly-namespaced child in the
   error element. The application-specific element SHOULD supplement or
   further qualify a defined element. Thus, the <error/> element will
   contain two or three child elements:

   <iq type='error' id='some-id'>
     <error type='modify'>
       <bad-request xmlns='urn:ietf:params:xml:ns:xmpp-stanzas'/>
       <too-many-parameters xmlns='application-ns'/>
     </error>
   </iq>

   <message type='error' id='another-id'>
     <error type='modify'>
       <undefined-condition
             xmlns='urn:ietf:params:xml:ns:xmpp-stanzas'/>
       <text xml:lang='en'
             xmlns='urn:ietf:params:xml:ns:xmpp-stanzas'>
         Some special application diagnostic information...
       </text>
       <special-application-condition xmlns='application-ns'/>
     </error>
   </message>

10. Server Rules for Handling XML Stanzas

   Compliant server implementations MUST ensure in-order processing of
   XML stanzas between any two entities.

   Beyond the requirement for in-order processing, each server
   implementation will contain its own "delivery tree" for handling
   stanzas it receives. Such a tree determines whether a stanza needs
   to be routed to another domain, processed internally, or delivered to
   a resource associated with a connected node. The following rules
   apply:

10.1. No 'to' Address

   If the stanza possesses no 'to' attribute, the server SHOULD process
   it on behalf of the entity that sent it. Because all stanzas
   received from other servers MUST possess a 'to' attribute, this rule
   applies only to stanzas received from a registered entity (such as a
   client) that is connected to the server. If the server receives a
   presence stanza with no 'to' attribute, the server SHOULD broadcast
   it to the entities that are subscribed to the sending entity's
   presence, if applicable (the semantics of presence broadcast for
   instant messaging and presence applications are defined in
   [XMPP-IM]). If the server receives an IQ stanza of type "get" or
   "set" with no 'to' attribute and it understands the namespace that
   qualifies the content of the stanza, it MUST either process the
   stanza on behalf of the sending entity (where the meaning of
   "process" is determined by the semantics of the qualifying namespace)
   or return an error to the sending entity.

10.2. Foreign Domain

   If the hostname of the domain identifier portion of the JID contained
   in the 'to' attribute does not match one of the configured hostnames
   of the server itself or a subdomain thereof, the server SHOULD route
   the stanza to the foreign domain (subject to local service
   provisioning and security policies regarding inter-domain
   communication). There are two possible cases:

   A server-to-server stream already exists between the two domains: The
      sender's server routes the stanza to the authoritative server for
      the foreign domain over the existing stream

   There exists no server-to-server stream between the two domains: The
      sender's server (1) resolves the hostname of the foreign domain
      (as defined under Server-to-Server Communications (Section 14.4)),
      (2) negotiates a server-to-server stream between the two domains
      (as defined under Use of TLS (Section 5) and Use of SASL (Section

   6)), and (3) routes the stanza to the authoritative server for the
   foreign domain over the newly-established stream

   If routing to the recipient's server is unsuccessful, the sender's
   server MUST return an error to the sender; if the recipient's server
   can be contacted but delivery by the recipient's server to the
   recipient is unsuccessful, the recipient's server MUST return an
   error to the sender by way of the sender's server.

10.3. Subdomain

   If the hostname of the domain identifier portion of the JID contained
   in the 'to' attribute matches a subdomain of one of the configured
   hostnames of the server itself, the server MUST either process the
   stanza itself or route the stanza to a specialized service that is
   responsible for that subdomain (if the subdomain is configured), or
   return an error to the sender (if the subdomain is not configured).

10.4. Mere Domain or Specific Resource

   If the hostname of the domain identifier portion of the JID contained
   in the 'to' attribute matches a configured hostname of the server
   itself and the JID contained in the 'to' attribute is of the form
   <domain> or <domain/resource>, the server (or a defined resource
   thereof) MUST either process the stanza as appropriate for the stanza
   kind or return an error stanza to the sender.

10.5. Node in Same Domain

   If the hostname of the domain identifier portion of the JID contained
   in the 'to' attribute matches a configured hostname of the server
   itself and the JID contained in the 'to' attribute is of the form
   <node@domain> or <node@domain/resource>, the server SHOULD deliver
   the stanza to the intended recipient of the stanza as represented by
   the JID contained in the 'to' attribute. The following rules apply:

   1. If the JID contains a resource identifier (i.e., is of the form
       <node@domain/resource>) and there exists a connected resource
       that matches the full JID, the recipient's server SHOULD deliver
       the stanza to the stream or session that exactly matches the
       resource identifier.

   2. If the JID contains a resource identifier and there exists no
       connected resource that matches the full JID, the recipient's
       server SHOULD return a <service-unavailable/> stanza error to the
       sender.

   3. If the JID is of the form <node@domain> and there exists at least
       one connected resource for the node, the recipient's server
       SHOULD deliver the stanza to at least one of the connected
       resources, according to application-specific rules (a set of
       delivery rules for instant messaging and presence applications is
       defined in [XMPP-IM]).

11. XML Usage within XMPP

11.1. Restrictions

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

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

   o comments (as defined in Section 2.5 of [XML])

   o processing instructions (Section 2.6 therein)

   o internal or external DTD subsets (Section 2.8 therein)

   o internal or external entity references (Section 4.2 therein) with
      the exception of predefined entities (Section 4.6 therein)

   o character data or attribute values containing unescaped characters
      that map to the predefined entities (Section 4.6 therein); such
      characters MUST be escaped

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

11.2. XML Namespace Names and Prefixes

   XML Namespaces [XML-NAMES] 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 allowable XML to be structurally mixed
   with any data element within XMPP. Rules for XML namespace names and
   prefixes are defined in the following subsections.

11.2.1. Streams Namespace

   A streams namespace declaration is REQUIRED in all XML stream
   headers. The name of the streams namespace MUST be
   'http://etherx.jabber.org/streams'. The element names of the
   element and its and children MUST be
   qualified by the streams namespace prefix in all instances. An
   implementation SHOULD generate only the 'stream:' prefix for these
   elements, and for historical reasons MAY accept only the 'stream:'
   prefix.

11.2.2. Default Namespace

   A default namespace declaration is REQUIRED and is used in all XML
   streams in order to define the allowable first-level children of the
   root stream element. This namespace declaration MUST be the same for
   the initial stream and the response stream so that both streams are
   qualified consistently. The default namespace declaration applies to
   the stream and all stanzas sent within a stream (unless explicitly
   qualified by another namespace, or by the prefix of the streams
   namespace or the dialback namespace).

   A server implementation MUST support the following two default
   namespaces (for historical reasons, some 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

   A client implementation MUST support the 'jabber:client' default
   namespace, and for historical reasons MAY support only that default
   namespace.

   An implementation MUST NOT generate namespace prefixes for elements
   in the default namespace if the default namespace is 'jabber:client'
   or 'jabber:server'. An implementation SHOULD NOT generate namespace
   prefixes for elements qualified by content (as opposed to stream)
   namespaces other than 'jabber:client' and 'jabber:server'.

   Note: 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 qualified by the 'jabber:client' or 'jabber:server'
namespace, it MUST support the common attributes (Section 9.1) and
basic semantics (Section 9.2) of all three core stanza kinds
(message, presence, and IQ).

11.2.3. Dialback Namespace

A dialback namespace declaration is REQUIRED for all elements used in
server dialback (Section 8). The name of the dialback namespace MUST
be 'jabber:server:dialback'. All elements qualified by this
namespace MUST be prefixed. An implementation SHOULD generate only
the 'db:' prefix for such elements and MAY accept only the 'db:'
prefix.

11.3. Validation

Except as noted with regard to 'to' and 'from' addresses for stanzas
within the 'jabber:server' namespace, a server is not responsible for
validating the XML elements forwarded to a client or another server;
an implementation MAY choose to provide only validated data elements
but this is OPTIONAL (although an implementation MUST NOT accept XML
that is not well-formed). 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 OPTIONAL, and schemas are
included herein for descriptive purposes only.

11.4. Inclusion of Text Declaration

Implementations SHOULD send a text declaration before sending a
stream header. Applications MUST follow the rules in [XML] regarding
the circumstances under which a text declaration is included.

11.5. Character Encoding

Implementations MUST support the UTF-8 (RFC 3629 [UTF-8])
transformation of Universal Character Set (ISO/IEC 10646-1 [UCS2])
characters, as required by RFC 2277 [CHARSET]. Implementations MUST
NOT attempt to use any other encoding.

12. Core Compliance Requirements

This section summarizes the specific aspects of the Extensible
Messaging and Presence Protocol that MUST be supported by servers and
clients in order to be considered compliant implementations, as well
as additional protocol aspects that SHOULD be supported. For
compliance purposes, we draw a distinction between core protocols

(which MUST be supported by any server or client, regardless of the
specific application) and instant messaging protocols (which MUST be
supported only by instant messaging and presence applications built
on top of the core protocols). Compliance requirements that apply to
all servers and clients are specified in this section; compliance
requirements for instant messaging servers and clients are specified
in the corresponding section of [XMPP-IM].

12.1. Servers

In addition to all defined requirements with regard to security, XML
usage, and internationalization, a server MUST support the following
core protocols in order to be considered compliant:

o Application of the [NAMEPREP], Nodeprep (Appendix A), and
   Resourceprep (Appendix B) profiles of [STRINGPREP] to addresses
   (including ensuring that domain identifiers are internationalized
   domain names as defined in [IDNA])

o XML streams (Section 4), including Use of TLS (Section 5), Use of
   SASL (Section 6), and Resource Binding (Section 7)

o The basic semantics of the three defined stanza kinds (i.e.,
   , , and) as specified in stanza
   semantics (Section 9.2)

o Generation (and, where appropriate, handling) of error syntax and
   semantics related to streams, TLS, SASL, and XML stanzas

In addition, a server MAY support the following core protocol:

o Server dialback (Section 8)

12.2. Clients

A client MUST support the following core protocols in order to be
considered compliant:

o XML streams (Section 4), including Use of TLS (Section 5), Use of
   SASL (Section 6), and Resource Binding (Section 7)

o The basic semantics of the three defined stanza kinds (i.e.,
   , , and) as specified in stanza
   semantics (Section 9.2)

o Handling (and, where appropriate, generation) of error syntax and
   semantics related to streams, TLS, SASL, and XML stanzas

In addition, a client SHOULD support the following core protocols:

o Generation of addresses to which the [NAMEPREP], Nodeprep
   (Appendix A), and Resourceprep (Appendix B) profiles of
   [STRINGPREP] can be applied without failing

13. Internationalization Considerations

XML streams MUST be encoded in UTF-8 as specified under Character
Encoding (Section 11.5). As specified under Stream Attributes
(Section 4.4), an XML stream SHOULD include an 'xml:lang' attribute
that is treated as the default language for any XML character data
sent over the stream that is intended to be presented to a human
user. As specified under xml:lang (Section 9.1.5), an XML stanza
SHOULD include an 'xml:lang' attribute if the stanza contains XML
character data that is intended to be presented to a human user. A
server SHOULD apply the default 'xml:lang' attribute to stanzas it
routes or delivers on behalf of connected entities, and MUST NOT
modify or delete 'xml:lang' attributes from stanzas it receives from
other entities.

14. Security Considerations

14.1. High Security

For the purposes of XMPP communications (client-to-server and
server-to-server), the term "high security" refers to the use of
security technologies that provide both mutual authentication and
integrity-checking; in particular, when using certificate-based
authentication to provide high security, a chain-of-trust SHOULD be
established out-of-band, although a shared certificate authority
signing certificates could allow a previously unknown certificate to
establish trust in-band. See Section 14.2 below regarding
certificate validation procedures.

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

14.2. Certificate Validation

When an XMPP peer communicates with another peer securely, it MUST
validate the peer's certificate. There are three possible cases:

Case #1: The peer contains an End Entity certificate which appears to
   be certified by a chain of certificates terminating in a trust
   anchor (as described in Section 6.1 of [X509]).

Case #2: The peer certificate is certified by a Certificate Authority
   not known to the validating peer.

Case #3: The peer certificate is self-signed.

In Case #1, the validating peer MUST do one of two things:

1. Verify the peer certificate according to the rules of [X509].
    The certificate SHOULD then be checked against the expected
    identity of the peer following the rules described in [HTTP-TLS],
    except that a subjectAltName extension of type "xmpp" MUST be
    used as the identity if present. If one of these checks fails,
    user-oriented clients MUST either notify the user (clients MAY
    give the user the opportunity to continue with the connection in
    any case) or terminate the connection with a bad certificate
    error. Automated clients SHOULD terminate the connection (with a
    bad certificate error) and log the error to an appropriate audit
    log. Automated clients MAY provide a configuration setting that
    disables this check, but MUST provide a setting that enables it.

2. The peer SHOULD show the certificate to a user for approval,
    including the entire certificate chain. The peer MUST cache the
    certificate (or some non-forgeable representation such as a
    hash). In future connections, the peer MUST verify that the same
    certificate was presented and MUST notify the user if it has
    changed.

In Case #2 and Case #3, implementations SHOULD act as in (2) above.

14.3. Client-to-Server Communications

A compliant client implementation MUST support both TLS and SASL for
connections to a server.

The TLS protocol for encrypting XML streams (defined under Use of TLS
(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 Use
of SASL (Section 6)) provides a reliable mechanism for validating
that a client connecting to a server is who it claims to be.

Client-to-server communications MUST NOT proceed until the DNS
hostname asserted by the server has been resolved. Such resolutions
SHOULD first attempt to resolve the hostname using an [SRV] Service
of "xmpp-client" and Proto of "tcp", resulting in resource records
such as "_xmpp-client._tcp.example.com." (the use of the string

"xmpp-client" for the service identifier is consistent with the IANA
registration). If the SRV lookup fails, the fallback is a normal
IPv4/IPv6 address record resolution to determine the IP address,
using the "xmpp-client" port of 5222, registered with the IANA.

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

14.4. Server-to-Server Communications

A compliant server implementation MUST support both TLS and SASL for
inter-domain communications. For historical reasons, a compliant
implementation SHOULD also support Server Dialback (Section 8).

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.

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

Inter-domain connections MUST NOT proceed until the DNS hostnames
asserted by the servers have been resolved. Such resolutions MUST
first attempt to resolve the hostname using an [SRV] Service of
"xmpp-server" and Proto of "tcp", resulting in resource records such
as "_xmpp-server._tcp.example.com." (the use of the string
"xmpp-server" for the service identifier is consistent with the IANA
registration; note well that the "xmpp-server" service identifier
supersedes the earlier use of a "jabber" service identifier, since
the earlier usage did not conform to [SRV]; implementations desiring
to be backward compatible should continue to look for or answer to
the "jabber" service identifier as well). If the SRV lookup fails,
the fallback is a normal IPv4/IPv6 address record resolution to
determine the IP address, using the "xmpp-server" port 5269,
registered with the IANA.

Server dialback helps protect against domain spoofing, thus making it
more difficult to spoof XML stanzas. It is not a mechanism for
authenticating, securing, or encrypting streams between servers as is
done via SASL and TLS, and results in weak verification of server
identities only. Furthermore, it is susceptible to DNS poisoning
attacks unless DNSSec [DNSSEC] is used, and even if the DNS

   information is accurate, dialback cannot protect from attacks where
   the attacker is capable of hijacking the IP address of the remote
   domain. Domains requiring robust security SHOULD use TLS and SASL.
   If SASL is used for server-to-server authentication, dialback SHOULD
   NOT be used since it is unnecessary.

14.5. Order of Layers

   The order of layers in which protocols MUST be stacked is as follows:

   1. TCP
   2. TLS
   3. SASL
   4. XMPP

   The rationale for this order is that [TCP] is the base connection
   layer used by all of the protocols stacked on top of TCP, [TLS] is
   often provided at the operating system layer, [SASL] is often
   provided at the application layer, and XMPP is the application
   itself.

14.6. Lack of SASL Channel Binding to TLS

   The SASL framework does not provide a mechanism to bind SASL
   authentication to a security layer providing confidentiality and
   integrity protection that was negotiated at a lower layer. This lack
   of a "channel binding" prevents SASL from being able to verify that
   the source and destination end points to which the lower layer's
   security is bound are equivalent to the end points that SASL is
   authenticating. If the end points are not identical, the lower
   layer's security cannot be trusted to protect data transmitted
   between the SASL authenticated entities. In such a situation, a SASL
   security layer should be negotiated that effectively ignores the
   presence of the lower layer security.

14.7. 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 plus SASL EXTERNAL(using the
      TLS_RSA_WITH_3DES_EDE_CBC_SHA cipher supporting client-side
      certificates)

14.8. Firewalls

   Communications using XMPP normally occur over [TCP] connections on
   port 5222 (client-to-server) or port 5269 (server-to-server), as
   registered with the IANA (see IANA Considerations (Section 15)). Use
   of these well-known ports allows administrators to easily enable or
   disable XMPP activity through existing and commonly-deployed
   firewalls.

14.9. Use of base64 in SASL

   Both the client and the server MUST verify any [BASE64] data received
   during SASL negotiation. An implementation MUST reject (not ignore)
   any characters that are not explicitly allowed by the base64
   alphabet; this helps to guard against creation of a covert channel
   that could be used to "leak" information. An implementation MUST NOT
   break on invalid input and MUST reject any sequence of base64
   characters containing the pad ('=') character if that character is
   included as something other than the last character of the data
   (e.g., "=AAA" or "BBBB=CCC"); this helps to guard against buffer
   overflow attacks and other attacks on the implementation. Base 64
   encoding visually hides otherwise easily recognized information, such
   as passwords, but does not provide any computational confidentiality.
   Base 64 encoding MUST follow the definition in Section 3 of RFC 3548
   [BASE64].

14.10. Stringprep Profiles

   XMPP makes use of the [NAMEPREP] profile of [STRINGPREP] for the
   processing of domain identifiers; for security considerations related
   to Nameprep, refer to the appropriate section of [NAMEPREP].

   In addition, XMPP defines two profiles of [STRINGPREP]: Nodeprep
   (Appendix A) for node identifiers and Resourceprep (Appendix B) for
   resource identifiers.

   The Unicode and ISO/IEC 10646 repertoires have many characters that
   look similar. In many cases, users of security protocols might do
   visual matching, such as when comparing the names of trusted third
   parties. Because it is impossible to map similar-looking characters
   without a great deal of context, such as knowing the fonts used,
   stringprep does nothing to map similar-looking characters together,
   nor to prohibit some characters because they look like others.

   A node identifier can be employed as one part of an entity's address
   in XMPP. One common usage is as the username of an instant messaging
   user; another is as the name of a multi-user chat room; many other
   kinds of entities could use node identifiers as part of their

addresses. The security of such services could be compromised based
on different interpretations of the internationalized node
identifier; for example, a user entering a single internationalized
node identifier could access another user's account information, or a
user could gain access to an otherwise restricted chat room or
service.

A resource identifier can be employed as one part of an entity's
address in XMPP. One common usage is as the name for an instant
messaging user's connected resource (active session); another is as
the nickname of a user in a multi-user chat room; many other kinds of
entities could use resource identifiers as part of their addresses.
The security of such services could be compromised based on different
interpretations of the internationalized resource identifier; for
example, a user could attempt to initiate multiple sessions with the
same name, or a user could send a message to someone other than the
intended recipient in a multi-user chat room.

15.   IANA Considerations

15.1.   XML Namespace Name for TLS Data

A URN sub-namespace for TLS-related data in the Extensible Messaging
and Presence Protocol (XMPP) is defined as follows. (This namespace
name adheres to the format defined in The IETF XML Registry
[XML-REG].)

    URI: urn:ietf:params:xml:ns:xmpp-tls
    Specification: RFC 3920
    Description: This is the XML namespace name for TLS-related data in
       the Extensible Messaging and Presence Protocol (XMPP) as defined
       by RFC 3920.
    Registrant Contact: IETF, XMPP Working Group, <xmppwg@jabber.org>

15.2.   XML Namespace Name for SASL Data

A URN sub-namespace for SASL-related data in the Extensible Messaging
and Presence Protocol (XMPP) is defined as follows. (This namespace
name adheres to the format defined in [XML-REG].)

    URI: urn:ietf:params:xml:ns:xmpp-sasl
    Specification: RFC 3920
    Description: This is the XML namespace name for SASL-related data in
       the Extensible Messaging and Presence Protocol (XMPP) as defined
       by RFC 3920.
    Registrant Contact: IETF, XMPP Working Group, <xmppwg@jabber.org>

15.3. XML Namespace Name for Stream Errors

   A URN sub-namespace for stream-related error data in the Extensible
   Messaging and Presence Protocol (XMPP) is defined as follows. (This
   namespace name adheres to the format defined in [XML-REG].)

   URI: urn:ietf:params:xml:ns:xmpp-streams
   Specification: RFC 3920
   Description: This is the XML namespace name for stream-related error
      data in the Extensible Messaging and Presence Protocol (XMPP) as
      defined by RFC 3920.
   Registrant Contact: IETF, XMPP Working Group, <xmppwg@jabber.org>

15.4. XML Namespace Name for Resource Binding

   A URN sub-namespace for resource binding in the Extensible Messaging
   and Presence Protocol (XMPP) is defined as follows. (This namespace
   name adheres to the format defined in [XML-REG].)

   URI: urn:ietf:params:xml:ns:xmpp-bind
   Specification: RFC 3920
   Description: This is the XML namespace name for resource binding in
      the Extensible Messaging and Presence Protocol (XMPP) as defined
      by RFC 3920.
   Registrant Contact: IETF, XMPP Working Group, <xmppwg@jabber.org>

15.5. XML Namespace Name for Stanza Errors

   A URN sub-namespace for stanza-related error data in the Extensible
   Messaging and Presence Protocol (XMPP) is defined as follows. (This
   namespace name adheres to the format defined in [XML-REG].)

   URI: urn:ietf:params:xml:ns:xmpp-stanzas
   Specification: RFC 3920
   Description: This is the XML namespace name for stanza-related error
      data in the Extensible Messaging and Presence Protocol (XMPP) as
      defined by RFC 3920.
   Registrant Contact: IETF, XMPP Working Group, <xmppwg@jabber.org>

15.6. Nodeprep Profile of Stringprep

   The Nodeprep profile of stringprep is defined under Nodeprep
   (Appendix A). The IANA has registered Nodeprep in the stringprep
   profile registry.

   Name of this profile:

      Nodeprep

RFC in which the profile is defined:

RFC 3920

Indicator whether or not this is the newest version of the profile:

This is the first version of Nodeprep

15.7. Resourceprep Profile of Stringprep

The Resourceprep profile of stringprep is defined under Resourceprep
(Appendix B). The IANA has registered Resourceprep in the stringprep
profile registry.

Name of this profile:

Resourceprep

RFC in which the profile is defined:

RFC 3920

Indicator whether or not this is the newest version of the profile:

This is the first version of Resourceprep

15.8. GSSAPI Service Name

The IANA has registered "xmpp" as a GSSAPI [GSS-API] service name, as
defined under SASL Definition (Section 6.3).

15.9. Port Numbers

The IANA has registered "xmpp-client" and "xmpp-server" as keywords
for [TCP] ports 5222 and 5269 respectively.

These ports SHOULD be used for client-to-server and server-to-server
communications respectively, but their use is OPTIONAL.

16. References

16.1. Normative References

   [ABNF]   Crocker, D. and P. Overell, "Augmented BNF for Syntax
                 Specifications: ABNF", RFC 2234, November 1997.

   [BASE64]   Josefsson, S., "The Base16, Base32, and Base64 Data
                 Encodings", RFC 3548, July 2003.

   [CHARSET]   Alvestrand, H., "IETF Policy on Character Sets and
                   Languages", BCP 18, RFC 2277, January 1998.

   [DIGEST-MD5] Leach, P. and C. Newman, "Using Digest Authentication as
                   a SASL Mechanism", RFC 2831, May 2000.

   [DNS]   Mockapetris, P., "Domain names - implementation and
                   specification", STD 13, RFC 1035, November 1987.

   [GSS-API]   Linn, J., "Generic Security Service Application Program
                   Interface Version 2, Update 1", RFC 2743, January 2000.

   [HTTP-TLS]   Rescorla, E., "HTTP Over TLS", RFC 2818, May 2000.

   [IDNA]   Faltstrom, P., Hoffman, P., and A. Costello,
                   "Internationalizing Domain Names in Applications
                   (IDNA)", RFC 3490, March 2003.

   [IPv6]   Hinden, R. and S. Deering, "Internet Protocol Version 6
                   (IPv6) Addressing Architecture", RFC 3513, April 2003.

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

   [NAMEPREP]   Hoffman, P. and M. Blanchet, "Nameprep: A Stringprep
                   Profile for Internationalized Domain Names (IDN)", RFC
                   3491, March 2003.

   [RANDOM]   Eastlake 3rd, D., Crocker, S., and J. Schiller,
                   "Randomness Recommendations for Security", RFC 1750,
                   December 1994.

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

   [SRV]   Gulbrandsen, A., Vixie, P., and L. Esibov, "A DNS RR for
                   specifying the location of services (DNS SRV)", RFC
                   2782, February 2000.

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

   [TCP]   Postel, J., "Transmission Control Protocol", STD 7, RFC
                   793, September 1981.

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

   [TLS]   Dierks, T. and C. Allen, "The TLS Protocol Version 1.0",
                  RFC 2246, January 1999.

   [UCS2]   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.

   [UTF-8]   Yergeau, F., "UTF-8, a transformation format of ISO
                  10646", STD 63, RFC 3629, November 2003.

   [X509]   Housley, R., Polk, W., Ford, W., and D. Solo, "Internet
                  X.509 Public Key Infrastructure Certificate and
                  Certificate Revocation List (CRL) Profile", RFC 3280,
                  April 2002.

   [XML]   Bray, T., Paoli, J., Sperberg-McQueen, C., and E. Maler,
                  "Extensible Markup Language (XML) 1.0 (2nd ed)", W3C
                  REC-xml, October 2000, <http://www.w3.org/TR/REC-xml>.

   [XML-NAMES]   Bray, T., Hollander, D., and A. Layman, "Namespaces in
                  XML", W3C REC-xml-names, January 1999,
                  <http://www.w3.org/TR/REC-xml-names>.

16.2. Informative References

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

   [ASN.1]   CCITT, "Recommendation X.208: Specification of Abstract
                  Syntax Notation One (ASN.1)", 1988.

   [DNSSEC]   Eastlake 3rd, D., "Domain Name System Security
                  Extensions", RFC 2535, March 1999.

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

   [IMAP]   Crispin, M., "INTERNET MESSAGE ACCESS PROTOCOL - VERSION
                  4rev1", RFC 3501, March 2003.

   [IMP-REQS]   Day, M., Aggarwal, S., Mohr, G., and J. Vincent,
                 "Instant Messaging / Presence Protocol Requirements",
                 RFC 2779, February 2000.

   [IRC]   Oikarinen, J. and D. Reed, "Internet Relay Chat
                 Protocol", RFC 1459, May 1993.

   [JEP-0029]   Kaes, C., "Definition of Jabber Identifiers (JIDs)", JSF
                 JEP 0029, October 2003.

   [JEP-0078]   Saint-Andre, P., "Non-SASL Authentication", JSF JEP
                 0078, July 2004.

   [JEP-0086]   Norris, R. and P. Saint-Andre, "Error Condition
                 Mappings", JSF JEP 0086, February 2004.

   [JSF]   Jabber Software Foundation, "Jabber Software
                 Foundation", <http://www.jabber.org/>.

   [POP3]   Myers, J. and M. Rose, "Post Office Protocol - Version
                 3", STD 53, RFC 1939, May 1996.

   [SIMPLE]   SIMPLE Working Group, "SIMPLE WG",
                 <http://www.ietf.org/html.charters/simple-charter.html>.

   [SMTP]   Klensin, J., "Simple Mail Transfer Protocol", RFC 2821,
                 April 2001.

   [URI]   Berners-Lee, T., Fielding, R., and L. Masinter, "Uniform
                 Resource Identifiers (URI): Generic Syntax", RFC 2396,
                 August 1998.

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

   [XML-REG]   Mealling, M., "The IETF XML Registry", BCP 81, RFC 3688,
                 January 2004.

   [XMPP-IM]   Saint-Andre, P., Ed., "Extensible Messaging and Presence
                 Protocol (XMPP): Instant Messaging and Presence", RFC
                 3921, October 2004.

Appendix A. Nodeprep

A.1. Introduction

   This appendix defines the "Nodeprep" profile of [STRINGPREP]. As
   such, it specifies processing rules that will enable users to enter
   internationalized node identifiers in the Extensible Messaging and
   Presence Protocol (XMPP) and have the highest chance of getting the
   content of the strings correct. (An XMPP node identifier is the
   optional portion of an XMPP address that precedes a domain identifier
   and the '@' separator; it is often but not exclusively associated
   with an instant messaging username.) These processing rules are
   intended only for XMPP node identifiers and are not intended for
   arbitrary text or any other aspect of an XMPP address.

   This profile defines the following, as required by [STRINGPREP]:

   o The intended applicability of the profile: internationalized node
      identifiers within XMPP
   o The character repertoire that is the input and output to
      stringprep: Unicode 3.2, specified in Section 2 of this Appendix
   o The mappings used: specified in Section 3
   o The Unicode normalization used: specified in Section 4
   o The characters that are prohibited as output: specified in Section
      5
   o Bidirectional character handling: specified in Section 6

A.2. Character Repertoire

   This profile uses Unicode 3.2 with the list of unassigned code points
   being Table A.1, both defined in Appendix A of [STRINGPREP].

A.3. Mapping

   This profile specifies mapping using the following tables from
   [STRINGPREP]:

      Table B.1
      Table B.2

A.4. Normalization

   This profile specifies the use of Unicode normalization form KC, as
   described in [STRINGPREP].

A.5. Prohibited Output

   This profile specifies the prohibition of using the following tables
   from [STRINGPREP].

      Table C.1.1
      Table C.1.2
      Table C.2.1
      Table C.2.2
      Table C.3
      Table C.4
      Table C.5
      Table C.6
      Table C.7
      Table C.8
      Table C.9

   In addition, the following Unicode characters are also prohibited:

      #x22 (")
      #x26 (&)
      #x27 (')
      #x2F (/)
      #x3A (:)
      #x3C (<)
      #x3E (>)
      #x40 (@)

A.6. Bidirectional Characters

   This profile specifies the checking of bidirectional strings, as
   described in Section 6 of [STRINGPREP].

Appendix B. Resourceprep

B.1. Introduction

   This appendix defines the "Resourceprep" profile of [STRINGPREP]. As
   such, it specifies processing rules that will enable users to enter
   internationalized resource identifiers in the Extensible Messaging
   and Presence Protocol (XMPP) and have the highest chance of getting
   the content of the strings correct. (An XMPP resource identifier is
   the optional portion of an XMPP address that follows a domain
   identifier and the '/' separator; it is often but not exclusively
   associated with an instant messaging session name.) These processing
   rules are intended only for XMPP resource identifiers and are not
   intended for arbitrary text or any other aspect of an XMPP address.

This profile defines the following, as required by [STRINGPREP]:

o The intended applicability of the profile: internationalized
   resource identifiers within XMPP

o The character repertoire that is the input and output to
   stringprep: Unicode 3.2, specified in Section 2 of this Appendix

o The mappings used: specified in Section 3

o The Unicode normalization used: specified in Section 4

o The characters that are prohibited as output: specified in Section
   5

o Bidirectional character handling: specified in Section 6

B.2. Character Repertoire

   This profile uses Unicode 3.2 with the list of unassigned code points
   being Table A.1, both defined in Appendix A of [STRINGPREP].

B.3. Mapping

   This profile specifies mapping using the following tables from
   [STRINGPREP]:

      Table B.1

B.4. Normalization

   This profile specifies using Unicode normalization form KC, as
   described in [STRINGPREP].

B.5. Prohibited Output

   This profile specifies prohibiting use of the following tables from
   [STRINGPREP].

      Table C.1.2
      Table C.2.1
      Table C.2.2
      Table C.3
      Table C.4
      Table C.5
      Table C.6
      Table C.7
      Table C.8
      Table C.9

B.6. Bidirectional Characters

   This profile specifies checking bidirectional strings as described in
   Section 6 of [STRINGPREP].

Appendix C. XML Schemas

   The following XML schemas are descriptive, not normative. For
   schemas defining the 'jabber:client' and 'jabber:server' namespaces,
   refer to [XMPP-IM].

C.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='unqualified'>

     <xs:element name='stream'>
       <xs:complexType>
         <xs:sequence xmlns:client='jabber:client'
                      xmlns:server='jabber:server'
                      xmlns:db='jabber:server:dialback'>
           <xs:element ref='features' minOccurs='0' maxOccurs='1'/>
           <xs:any namespace='urn:ietf:params:xml:ns:xmpp-tls'
                   minOccurs='0'
                   maxOccurs='unbounded'/>
           <xs:any namespace='urn:ietf:params:xml:ns:xmpp-sasl'
                   minOccurs='0'

```
                   maxOccurs='unbounded'/>
          <xs:choice minOccurs='0' maxOccurs='1'>
            <xs:choice minOccurs='0' maxOccurs='unbounded'>
              <xs:element ref='client:message'/>
              <xs:element ref='client:presence'/>
              <xs:element ref='client:iq'/>
            </xs:choice>
            <xs:choice minOccurs='0' maxOccurs='unbounded'>
              <xs:element ref='server:message'/>
              <xs:element ref='server:presence'/>
              <xs:element ref='server:iq'/>
              <xs:element ref='db:result'/>
              <xs:element ref='db:verify'/>
            </xs:choice>
          </xs:choice>
          <xs:element ref='error' minOccurs='0' maxOccurs='1'/>
        </xs:sequence>
        <xs:attribute name='from' type='xs:string' use='optional'/>
        <xs:attribute name='id' type='xs:NMTOKEN' use='optional'/>
        <xs:attribute name='to' type='xs:string' use='optional'/>
        <xs:attribute name='version' type='xs:decimal' use='optional'/>
        <xs:attribute ref='xml:lang' use='optional'/>
      </xs:complexType>
    </xs:element>

    <xs:element name='features'>
      <xs:complexType>
        <xs:all xmlns:tls='urn:ietf:params:xml:ns:xmpp-tls'
                xmlns:sasl='urn:ietf:params:xml:ns:xmpp-sasl'
                xmlns:bind='urn:ietf:params:xml:ns:xmpp-bind'
                xmlns:sess='urn:ietf:params:xml:ns:xmpp-session'>
          <xs:element ref='tls:starttls' minOccurs='0'/>
          <xs:element ref='sasl:mechanisms' minOccurs='0'/>
          <xs:element ref='bind:bind' minOccurs='0'/>
          <xs:element ref='sess:session' minOccurs='0'/>
        </xs:all>
      </xs:complexType>
    </xs:element>

    <xs:element name='error'>
      <xs:complexType>
        <xs:sequence  xmlns:err='urn:ietf:params:xml:ns:xmpp-streams'>
          <xs:group    ref='err:streamErrorGroup'/>
          <xs:element ref='err:text'
                      minOccurs='0'
                      maxOccurs='1'/>
        </xs:sequence>
      </xs:complexType>
```

```
        </xs:element>

    </xs:schema>
```

C.2. Stream error namespace

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

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

      <xs:element name='bad-format' type='empty'/>
      <xs:element name='bad-namespace-prefix' type='empty'/>
      <xs:element name='conflict' type='empty'/>
      <xs:element name='connection-timeout' type='empty'/>
      <xs:element name='host-gone' type='empty'/>
      <xs:element name='host-unknown' type='empty'/>
      <xs:element name='improper-addressing' type='empty'/>
      <xs:element name='internal-server-error' type='empty'/>
      <xs:element name='invalid-from' type='empty'/>
      <xs:element name='invalid-id' type='empty'/>
      <xs:element name='invalid-namespace' type='empty'/>
      <xs:element name='invalid-xml' type='empty'/>
      <xs:element name='not-authorized' type='empty'/>
      <xs:element name='policy-violation' type='empty'/>
      <xs:element name='remote-connection-failed' type='empty'/>
      <xs:element name='resource-constraint' type='empty'/>
      <xs:element name='restricted-xml' type='empty'/>
      <xs:element name='see-other-host' type='xs:string'/>
      <xs:element name='system-shutdown' type='empty'/>
      <xs:element name='undefined-condition' type='empty'/>
      <xs:element name='unsupported-encoding' type='empty'/>
      <xs:element name='unsupported-stanza-type' type='empty'/>
      <xs:element name='unsupported-version' type='empty'/>
      <xs:element name='xml-not-well-formed' type='empty'/>

      <xs:group name='streamErrorGroup'>
        <xs:choice>
          <xs:element ref='bad-format'/>
          <xs:element ref='bad-namespace-prefix'/>
          <xs:element ref='conflict'/>
          <xs:element ref='connection-timeout'/>
          <xs:element ref='host-gone'/>
          <xs:element ref='host-unknown'/>
          <xs:element ref='improper-addressing'/>
```

```
            <xs:element ref='internal-server-error'/>
            <xs:element ref='invalid-from'/>
            <xs:element ref='invalid-id'/>
            <xs:element ref='invalid-namespace'/>
            <xs:element ref='invalid-xml'/>
            <xs:element ref='not-authorized'/>
            <xs:element ref='policy-violation'/>
            <xs:element ref='remote-connection-failed'/>
            <xs:element ref='resource-constraint'/>
            <xs:element ref='restricted-xml'/>
            <xs:element ref='see-other-host'/>
            <xs:element ref='system-shutdown'/>
            <xs:element ref='undefined-condition'/>
            <xs:element ref='unsupported-encoding'/>
            <xs:element ref='unsupported-stanza-type'/>
            <xs:element ref='unsupported-version'/>
            <xs:element ref='xml-not-well-formed'/>
          </xs:choice>
       </xs:group>

       <xs:element name='text'>
         <xs:complexType>
           <xs:simpleContent>
             <xs:extension base='xs:string'>
               <xs:attribute ref='xml:lang' use='optional'/>
             </xs:extension>
           </xs:simpleContent>
         </xs:complexType>
       </xs:element>

       <xs:simpleType name='empty'>
         <xs:restriction base='xs:string'>
           <xs:enumeration value=''/>
         </xs:restriction>
       </xs:simpleType>

     </xs:schema>
```

C.3. TLS namespace

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

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

```
   <xs:element name='starttls'>
     <xs:complexType>
       <xs:sequence>
         <xs:element
             name='required'
             minOccurs='0'
             maxOccurs='1'
             type='empty'/>
       </xs:sequence>
     </xs:complexType>
   </xs:element>

   <xs:element name='proceed' type='empty'/>
   <xs:element name='failure' type='empty'/>

   <xs:simpleType name='empty'>
     <xs:restriction base='xs:string'>
       <xs:enumeration value=''/>
     </xs:restriction>
   </xs:simpleType>

 </xs:schema>
```

C.4.   SASL namespace

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

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

   <xs:element name='mechanisms'>
     <xs:complexType>
       <xs:sequence>
         <xs:element name='mechanism'
                     maxOccurs='unbounded'
                     type='xs:string'/>
       </xs:sequence>
     </xs:complexType>
   </xs:element>

   <xs:element name='auth'>
     <xs:complexType>
       <xs:simpleContent>
         <xs:extension base='empty'>
           <xs:attribute name='mechanism'
```

```
                            type='xs:string'
                            use='optional'/>
        </xs:extension>
      </xs:simpleContent>
    </xs:complexType>
  </xs:element>

  <xs:element name='challenge' type='xs:string'/>
  <xs:element name='response' type='xs:string'/>
  <xs:element name='abort' type='empty'/>
  <xs:element name='success' type='empty'/>

  <xs:element name='failure'>
    <xs:complexType>
      <xs:choice minOccurs='0'>
        <xs:element name='aborted' type='empty'/>
        <xs:element name='incorrect-encoding' type='empty'/>
        <xs:element name='invalid-authzid' type='empty'/>
        <xs:element name='invalid-mechanism' type='empty'/>
        <xs:element name='mechanism-too-weak' type='empty'/>
        <xs:element name='not-authorized' type='empty'/>
        <xs:element name='temporary-auth-failure' type='empty'/>
      </xs:choice>
    </xs:complexType>
  </xs:element>
  <xs:simpleType name='empty'>
    <xs:restriction base='xs:string'>
      <xs:enumeration value=''/>
    </xs:restriction>
  </xs:simpleType>

</xs:schema>
```

C.5. Resource binding namespace

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

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

  <xs:element name='bind'>
    <xs:complexType>
      <xs:choice minOccurs='0' maxOccurs='1'>
        <xs:element name='resource' type='xs:string'/>
        <xs:element name='jid' type='xs:string'/>
```

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

      </xs:schema>

C.6.  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:token'>
              <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:token'>
              <xs:attribute name='from' type='xs:string' use='required'/>
              <xs:attribute name='id' type='xs:NMTOKEN' 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:schema>

C.7.  Stanza error namespace

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

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

          <xs:element name='bad-request' type='empty'/>
          <xs:element name='conflict' type='empty'/>
          <xs:element name='feature-not-implemented' type='empty'/>
          <xs:element name='forbidden' type='empty'/>
          <xs:element name='gone' type='xs:string'/>
          <xs:element name='internal-server-error' type='empty'/>
          <xs:element name='item-not-found' type='empty'/>
          <xs:element name='jid-malformed' type='empty'/>
          <xs:element name='not-acceptable' type='empty'/>
          <xs:element name='not-allowed' type='empty'/>
          <xs:element name='payment-required' type='empty'/>
          <xs:element name='recipient-unavailable' type='empty'/>
          <xs:element name='redirect' type='xs:string'/>
          <xs:element name='registration-required' type='empty'/>
          <xs:element name='remote-server-not-found' type='empty'/>
          <xs:element name='remote-server-timeout' type='empty'/>
          <xs:element name='resource-constraint' type='empty'/>
          <xs:element name='service-unavailable' type='empty'/>
          <xs:element name='subscription-required' type='empty'/>
          <xs:element name='undefined-condition' type='empty'/>
          <xs:element name='unexpected-request' type='empty'/>

          <xs:group name='stanzaErrorGroup'>
            <xs:choice>
              <xs:element ref='bad-request'/>
              <xs:element ref='conflict'/>
              <xs:element ref='feature-not-implemented'/>
              <xs:element ref='forbidden'/>
              <xs:element ref='gone'/>
```

```
      <xs:element ref='internal-server-error'/>
      <xs:element ref='item-not-found'/>
      <xs:element ref='jid-malformed'/>
      <xs:element ref='not-acceptable'/>
      <xs:element ref='not-allowed'/>
      <xs:element ref='payment-required'/>
      <xs:element ref='recipient-unavailable'/>
      <xs:element ref='redirect'/>
      <xs:element ref='registration-required'/>
      <xs:element ref='remote-server-not-found'/>
      <xs:element ref='remote-server-timeout'/>
      <xs:element ref='resource-constraint'/>
      <xs:element ref='service-unavailable'/>
      <xs:element ref='subscription-required'/>
      <xs:element ref='undefined-condition'/>
      <xs:element ref='unexpected-request'/>
    </xs:choice>
  </xs:group>

  <xs:element name='text'>
    <xs:complexType>
      <xs:simpleContent>
        <xs:extension base='xs:string'>
          <xs:attribute ref='xml:lang' use='optional'/>
        </xs:extension>
      </xs:simpleContent>
    </xs:complexType>
  </xs:element>

  <xs:simpleType name='empty'>
    <xs:restriction base='xs:string'>
      <xs:enumeration value=''/>
    </xs:restriction>
  </xs:simpleType>

</xs:schema>
```

Appendix D. Differences Between Core Jabber Protocols and XMPP

   This section is non-normative.

   XMPP has been adapted from the protocols originally developed in the
   Jabber open-source community, which can be thought of as "XMPP 0.9".
   Because there exists a large installed base of Jabber implementations
   and deployments, it may be helpful to specify the key differences
   between the relevant Jabber protocols and XMPP in order to expedite
   and encourage upgrades of those implementations and deployments to
   XMPP. This section summarizes the core differences, while the
   corresponding section of [XMPP-IM] summarizes the differences that
   relate specifically to instant messaging and presence applications.

D.1. Channel Encryption

   It was common practice in the Jabber community to use SSL for channel
   encryption on ports other than 5222 and 5269 (the convention is to
   use ports 5223 and 5270). XMPP uses TLS over the IANA-registered
   ports for channel encryption, as defined under Use of TLS (Section 5)
   herein.

D.2. Authentication

   The client-server authentication protocol developed in the Jabber
   community used a basic IQ interaction qualified by the
   'jabber:iq:auth' namespace (documentation of this protocol is
   contained in [JEP-0078], published by the Jabber Software Foundation
   [JSF]). XMPP uses SASL for authentication, as defined under Use of
   SASL (Section 6) herein.

   The Jabber community did not develop an authentication protocol for
   server-to-server communications, only the Server Dialback (Section 8)
   protocol to prevent server spoofing. XMPP supersedes Server Dialback
   with a true server-to-server authentication protocol, as defined
   under Use of SASL (Section 6) herein.

D.3. Resource Binding

   Resource binding in the Jabber community was handled via the
   'jabber:iq:auth' namespace (which was also used for client
   authentication with a server). XMPP defines a dedicated namespace
   for resource binding as well as the ability for a server to generate
   a resource identifier on behalf of a client, as defined under
   Resource Binding (Section 7).

D.4. JID Processing

   JID processing was somewhat loosely defined by the Jabber community
   (documentation of forbidden characters and case handling is contained
   in [JEP-0029], published by the Jabber Software Foundation [JSF]).
   XMPP specifies the use of [NAMEPREP] for domain identifiers and
   supplements Nameprep with two additional [STRINGPREP] profiles for
   JID processing: Nodeprep (Appendix A) for node identifiers and
   Resourceprep (Appendix B) for resource identifiers.

D.5. Error Handling

   Stream-related errors were handled in the Jabber community via XML
   character data text in a <stream:error/> element. In XMPP,
   stream-related errors are handled via an extensible mechanism defined
   under Stream Errors (Section 4.7) herein.

   Stanza-related errors were handled in the Jabber community via
   HTTP-style error codes. In XMPP, stanza-related errors are handled
   via an extensible mechanism defined under Stanza Errors (Section 9.3)
   herein. (Documentation of a mapping between Jabber and XMPP error
   handling mechanisms is contained in [JEP-0086], published by the
   Jabber Software Foundation [JSF].)

D.6. Internationalization

   Although use of UTF-8 has always been standard practice within the
   Jabber community, the community did not define mechanisms for
   specifying the language of human-readable text provided in XML
   character data. XMPP specifies the use of the 'xml:lang' attribute
   in such contexts, as defined under Stream Attributes (Section 4.4)
   and xml:lang (Section 9.1.5) herein.

D.7. Stream Version Attribute

   The Jabber community did not include a 'version' attribute in stream
   headers. XMPP specifies inclusion of that attribute as a way to
   signal support for the stream features (authentication, encryption,
   etc.) defined under Version Support (Section 4.4.1) herein.

Contributors

   Most of the core aspects of the Extensible Messaging and Presence
   Protocol were developed originally within the Jabber open-source
   community in 1999. This community was founded by Jeremie Miller, who
   released source code for the initial version of the jabber server in
   January 1999. Major early contributors to the base protocol also
   included Ryan Eatmon, Peter Millard, Thomas Muldowney, and Dave
   Smith. Work by the XMPP Working Group has concentrated especially on
   security and internationalization; in these areas, protocols for the
   use of TLS and SASL were originally contributed by Rob Norris, and
   stringprep profiles were originally contributed by Joe Hildebrand.
   The error code syntax was suggested by Lisa Dusseault.

Acknowledgements

   Thanks are due to a number of individuals in addition to the
   contributors listed. Although it is difficult to provide a complete
   list, the following individuals were particularly helpful in defining
   the protocols or in commenting on the specifications in this memo:
   Thomas Charron, Richard Dobson, Sam Hartman, Schuyler Heath, Jonathan
   Hogg, Cullen Jennings, Craig Kaes, Jacek Konieczny, Alexey Melnikov,
   Keith Minkler, Julian Missig, Pete Resnick, Marshall Rose, Alexey
   Shchepin, Jean-Louis Seguineau, Iain Shigeoka, Greg Troxel, and David
   Waite. Thanks also to members of the XMPP Working Group and the IETF
   community for comments and feedback provided throughout the life of
   this memo.

Author's Address

   Peter Saint-Andre (editor)
   Jabber Software Foundation

   EMail: stpeter@jabber.org

Full Copyright Statement

Intellectual Property

Acknowledgement