Internet Engineering Task Force (IETF)
Request for Comments: 6120
Obsoletes: 3920
Category: Standards Track
ISSN: 2070-1721

P. Saint-Andre
Cisco

March 2011

Extensible Messaging and Presence Protocol (XMPP): Core

Abstract

The Extensible Messaging and Presence Protocol (XMPP) is an application profile of the Extensible Markup Language (XML) that enables the near-real-time exchange of structured yet extensible data between any two or more network entities. This document defines XMPP's core protocol methods: setup and teardown of XML streams, channel encryption, authentication, error handling, and communication primitives for messaging, network availability ("presence"), and request-response interactions. This document obsoletes RFC 3920.

Status of This Memo

Copyright Notice

Table of Contents

1. Introduction

1.1. Overview

The Extensible Messaging and Presence Protocol (XMPP) is an application profile of the Extensible Markup Language [XML] that enables the near-real-time exchange of structured yet extensible data between any two or more network entities. This document defines XMPP's core protocol methods: setup and teardown of XML streams, channel encryption, authentication, error handling, and communication primitives for messaging, network availability ("presence"), and request-response interactions.

1.2. History

The basic syntax and semantics of XMPP were developed originally within the Jabber open-source community, mainly in 1999. In late 2002, the XMPP Working Group was chartered with developing an adaptation of the base Jabber protocol that would be suitable as an IETF instant messaging (IM) and presence technology in accordance with [IMP-REQS]. In October 2004, [RFC3920] and [RFC3921] were published, representing the most complete definition of XMPP at that time.

Since 2004 the Internet community has gained extensive implementation and deployment experience with XMPP, including formal interoperability testing carried out under the auspices of the XMPP Standards Foundation (XSF). This document incorporates comprehensive feedback from software developers and XMPP service providers, including a number of backward-compatible modifications summarized under Appendix D. As a result, this document reflects the rough consensus of the Internet community regarding the core features of XMPP 1.0, thus obsoleting RFC 3920.

1.3. Functional Summary

This non-normative section provides a developer-friendly, functional summary of XMPP; refer to the sections that follow for a normative definition of XMPP.

The purpose of XMPP is to enable the exchange of relatively small pieces of structured data (called "XML stanzas") over a network between any two (or more) entities. XMPP is typically implemented using a distributed client-server architecture, wherein a client needs to connect to a server in order to gain access to the network and thus be allowed to exchange XML stanzas with other entities (which can be associated with other servers). The process whereby a client connects to a server, exchanges XML stanzas, and ends the connection is:

1. Determine the IP address and port at which to connect, typically based on resolution of a fully qualified domain name (Section 3.2)

2. Open a Transmission Control Protocol [TCP] connection

3. Open an XML stream over TCP (Section 4.2)

4. Preferably negotiate Transport Layer Security [TLS] for channel encryption (Section 5)

5. Authenticate using a Simple Authentication and Security Layer [SASL] mechanism (Section 6)

6. Bind a resource to the stream (Section 7)

7. Exchange an unbounded number of XML stanzas with other entities on the network (Section 8)

8. Close the XML stream (Section 4.4)

9. Close the TCP connection

Within XMPP, one server can optionally connect to another server to enable inter-domain or inter-server communication. For this to happen, the two servers need to negotiate a connection between themselves and then exchange XML stanzas; the process for doing so is:

1. Determine the IP address and port at which to connect, typically based on resolution of a fully qualified domain name (Section 3.2)

2. Open a TCP connection

3. Open an XML stream (Section 4.2)

4. Preferably negotiate TLS for channel encryption (Section 5)

5. Authenticate using a Simple Authentication and Security Layer [SASL] mechanism (Section 6) *

6. Exchange an unbounded number of XML stanzas both directly for the servers and indirectly on behalf of entities associated with each server, such as connected clients (Section 8)

7. Close the XML stream (Section 4.4)

8. Close the TCP connection

* Interoperability Note: At the time of writing, most deployed servers still use the Server Dialback protocol [XEP-0220] to provide weak identity verification instead of using SASL with PKIX certificates to provide strong authentication, especially in cases where SASL negotiation would not result in strong authentication anyway (e.g., because TLS negotiation was not mandated by the peer server, or because the PKIX certificate presented by the peer server during TLS negotiation is self-signed and has not been previously accepted); for details, see [XEP-0220]. The solutions specified in this document offer a significantly stronger level of security (see also Section 13.6).

This document specifies how clients connect to servers and specifies the basic semantics of XML stanzas. However, this document does not define the "payloads" of the XML stanzas that might be exchanged once a connection is successfully established; instead, those payloads are defined by various XMPP extensions. For example, [XMPP-IM] defines extensions for basic instant messaging and presence functionality. In addition, various specifications produced in the XSF's XEP series [XEP-0001] define extensions for a wide range of applications.

1.4. Terminology

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

Certain security-related terms are to be understood in the sense defined in [SEC-TERMS]; such terms include, but are not limited to, "assurance", "attack", "authentication", "authorization", "certificate", "certification authority", "certification path", "confidentiality", "credential", "downgrade", "encryption", "hash value", "identity", "integrity", "signature", "self-signed certificate", "sign", "spoof", "tamper", "trust", "trust anchor", "validate", and "verify".

Certain terms related to certificates, domains, and application service identity are to be understood in the sense defined in [TLS-CERTS]; these include, but are not limited to, "PKIX certificate", "source domain", "derived domain", and the identifier types "CN-ID", "DNS-ID", and "SRV-ID".

Other security-related terms are to be understood in the sense defined in the referenced specifications (for example, "denial of service" as described in [DOS] or "end entity certificate" as described in [PKIX]).

The term "whitespace" is used to refer to any character or characters matching the "S" production from [XML], i.e., one or more instances of the SP, HTAB, CR, or LF rules defined in [ABNF].

The terms "localpart", "domainpart", and "resourcepart" are defined in [XMPP-ADDR].

The term "bare JID" refers to an XMPP address of the form <localpart@domainpart> (for an account at a server) or of the form <domainpart> (for a server).

The term "full JID" refers to an XMPP address of the form <localpart@domainpart/resourcepart> (for a particular authorized client or device associated with an account) or of the form <domainpart/resourcepart> (for a particular resource or script associated with a server).

The term "XML stream" (also "stream") is defined under Section 4.1.

The term "XML stanza" (also "stanza") is defined under Section 4.1. There are three kinds of stanzas: message, presence, and IQ (short for "Info/Query"). These communication primitives are defined under Sections 8.2.1, 8.2.2, and 8.2.3, respectively.

The term "originating entity" refers to the entity that first generates a stanza that is sent over an XMPP network (e.g., a connected client, an add-on service, or a server). The term "generated stanza" refers to the stanza so generated.

The term "input stream" designates an XML stream over which a server receives data from a connected client or remote server, and the term "output stream" designates an XML stream over which a server sends data to a connected client or remote server. The following terms designate some of the actions that a server can perform when processing data received over an input stream:

route: pass the data to a remote server for direct processing by the remote server or eventual delivery to a client associated with the remote server

deliver: pass the data to a connected client

ignore: discard the data without acting upon it or returning an error to the sender

When the term "ignore" is used with regard to client processing of data it receives, the phrase "without acting upon it" explicitly includes not presenting the data to a human user.

Following the "XML Notation" used in [IRI] to represent characters that cannot be rendered in ASCII-only documents, some examples in this document use the form "&#x...." as a notational device to represent [UNICODE] characters (e.g., the string "&#x0159;" stands for the Unicode character LATIN SMALL LETTER R WITH CARON); this form is definitely not to be sent over the wire in XMPP systems.

Consistent with the convention used in [URI] to represent Uniform Resource Identifiers, XMPP addresses in running text are enclosed between '<' and '>' (although natively they are not URIs).

In examples, lines have been wrapped for improved readability, "[...]" means elision, and the following prepended strings are used (these prepended strings are not to be sent over the wire):

- C: = a client
- E: = any XMPP entity

- I: = an initiating entity
- P: = a peer server
- R: = a receiving entity
- S: = a server
- S1: = server1
- S2: = server2

Readers need to be aware that the examples are not exhaustive and that, in examples for some protocol flows, the alternate steps shown would not necessarily be triggered by the exact data sent in the previous step; in all cases the protocol definitions specified in this document or in normatively referenced documents rule over any examples provided here. All examples are fictional and the information exchanged (e.g., usernames and passwords) does not represent any existing users or servers.

2. Architecture

XMPP provides a technology for the asynchronous, end-to-end exchange of structured data by means of direct, persistent XML streams among a distributed network of globally addressable, presence-aware clients and servers. Because this architectural style involves ubiquitous knowledge of network availability and a conceptually unlimited number of concurrent information transactions in the context of a given client-to-server or server-to-server session, we label it "Availability for Concurrent Transactions" (ACT) to distinguish it from the "Representational State Transfer" [REST] architectural style familiar from the World Wide Web. Although the architecture of XMPP is similar in important ways to that of email (see [EMAIL-ARCH]), it introduces several modifications to facilitate communication in close to real time. The salient features of this ACTive architectural style are as follows.

2.1. Global Addresses

As with email, XMPP uses globally unique addresses (based on the Domain Name System) in order to route and deliver messages over the network. All XMPP entities are addressable on the network, most particularly clients and servers but also various additional services that can be accessed by clients and servers. In general, server addresses are of the form <domainpart> (e.g., <im.example.com>), accounts hosted at a server are of the form <localpart@domainpart> (e.g., <juliet@im.example.com>, called a "bare JID"), and a particular connected device or resource that is currently authorized for interaction on behalf of an account is of the form <localpart@domainpart/resourcepart> (e.g., <juliet@im.example.com/balcony>, called a "full JID"). For historical reasons, XMPP addresses are often called Jabber IDs or JIDs. Because the formal specification of the XMPP address format depends on internationalization technologies that are in flux at the time of writing, the format is defined in [XMPP-ADDR] instead of this document. The terms "localpart", "domainpart", and "resourcepart" are defined more formally in [XMPP-ADDR].

2.2. Presence

XMPP includes the ability for an entity to advertise its network availability or "presence" to other entities. In XMPP, this availability for communication is signaled end-to-end by means of a dedicated communication primitive: the <presence/> stanza. Although knowledge of network availability is not strictly necessary for the exchange of XMPP messages, it facilitates real-time interaction because the originator of a message can know before initiating communication that the intended recipient is online and available. End-to-end presence is defined in [XMPP-IM].

2.3. Persistent Streams

Availability for communication is also built into each point-to-point "hop" through the use of persistent XML streams over long-lived TCP connections. These "always-on" client-to-server and server-to-server streams enable each party to push data to the other party at any time for immediate routing or delivery. XML streams are defined under Section 4.

2.4. Structured Data

The basic protocol data unit in XMPP is not an XML stream (which simply provides the transport for point-to-point communication) but an XML "stanza", which is essentially a fragment of XML that is sent over a stream. The root element of a stanza includes routing attributes (such as "from" and "to" addresses), and the child elements of the stanza contain a payload for delivery to the intended recipient. XML stanzas are defined under Section 8.

2.5. Distributed Network of Clients and Servers

In practice, XMPP consists of a network of clients and servers that inter-communicate (however, communication between any two given deployed servers is strictly discretionary and a matter of local service policy). Thus, for example, the user <juliet@im.example.com>

associated with the server <im.example.com> might be able to exchange messages, presence, and other structured data with the user <romeo@example.net> associated with the server <example.net>. This pattern is familiar from messaging protocols that make use of global addresses, such as the email network (see [SMTP] and [EMAIL-ARCH]). As a result, end-to-end communication in XMPP is logically peer-to-peer but physically client-to-server-to-server-to-client, as illustrated in the following diagram.

```
     example.net <--------------> im.example.com
        ^                                ^
        |                                |
        v                                v
   romeo@example.net          juliet@im.example.com
```

Figure 1: Distributed Client-Server Architecture

Informational Note: Architectures that employ XML streams (Section 4) and XML stanzas (Section 8) but that establish peer-to-peer connections directly between clients using technologies based on [LINKLOCAL] have been deployed, but such architectures are not defined in this specification and are best described as "XMPP-like"; for details, see [XEP-0174]. In addition, XML streams can be established end-to-end over any reliable transport, including extensions to XMPP itself; however, such methods are out of scope for this specification.

The following paragraphs describe the responsibilities of clients and servers on the network.

A client is an entity that establishes an XML stream with a server by authenticating using the credentials of a registered account (via SASL negotiation (Section 6)) and that then completes resource binding (Section 7) in order to enable delivery of XML stanzas between the server and the client over the negotiated stream. The client then uses XMPP to communicate with its server, other clients, and any other entities on the network, where the server is responsible for delivering stanzas to other connected clients at the same server or routing them to remote servers. Multiple clients can connect simultaneously to a server on behalf of the same registered account, where each client is differentiated by the resourcepart of an XMPP address (e.g., <juliet@im.example.com/balcony> vs. <juliet@im.example.com/chamber>), as defined under [XMPP-ADDR] and Section 7.

A server is an entity whose primary responsibilities are to:

- o Manage XML streams (Section 4) with connected clients and deliver XML stanzas (Section 8) to those clients over the negotiated streams; this includes responsibility for ensuring that a client authenticates with the server before being granted access to the XMPP network.

- o Subject to local service policies on server-to-server communication, manage XML streams (Section 4) with remote servers and route XML stanzas (Section 8) to those servers over the negotiated streams.

Depending on the application, the secondary responsibilities of an XMPP server can include:

- o Storing data that is used by clients (e.g., contact lists for users of XMPP-based instant messaging and presence applications as defined in [XMPP-IM]); in this case, the relevant XML stanza is handled directly by the server itself on behalf of the client and is not routed to a remote server or delivered to a connected client.

- o Hosting add-on services that also use XMPP as the basis for communication but that provide additional functionality beyond that defined in this document or in [XMPP-IM]; examples include multi-user conferencing services as specified in [XEP-0045] and publish-subscribe services as specified in [XEP-0060].

3. TCP Binding

3.1. Scope

As XMPP is defined in this specification, an initiating entity (client or server) MUST open a Transmission Control Protocol [TCP] connection to the receiving entity (server) before it negotiates XML streams with the receiving entity. The parties then maintain that TCP connection for as long as the XML streams are in use. The rules specified in the following sections apply to the TCP binding.

> Informational Note: There is no necessary coupling of XML streams to TCP, and other transports are possible. For example, two entities could connect to each other by means of [HTTP] as specified in [XEP-0124] and [XEP-0206]. However, this specification defines only a binding of XMPP to TCP.

3.2. Resolution of Fully Qualified Domain Names

Because XML streams are sent over TCP, the initiating entity needs to determine the IPv4 or IPv6 address (and port) of the receiving entity before it can attempt to open an XML stream. Typically this is done by resolving the receiving entity's fully qualified domain name or FQDN (see [DNS-CONCEPTS]).

3.2.1. Preferred Process: SRV Lookup

The preferred process for FQDN resolution is to use [DNS-SRV] records as follows:

1. The initiating entity constructs a DNS SRV query whose inputs are:

   * a Service of "xmpp-client" (for client-to-server connections) or "xmpp-server" (for server-to-server connections)

   * a Proto of "tcp"

   * a Name corresponding to the "origin domain" [TLS-CERTS] of the XMPP service to which the initiating entity wishes to connect (e.g., "example.net" or "im.example.com")

2. The result is a query such as "_xmpp-client._tcp.example.net." or "_xmpp-server._tcp.im.example.com.".

3. If a response is received, it will contain one or more combinations of a port and FDQN, each of which is weighted and prioritized as described in [DNS-SRV]. (However, if the result of the SRV lookup is a single resource record with a Target of ".", i.e., the root domain, then the initiating entity MUST abort SRV processing at this point because according to [DNS-SRV] such a Target "means that the service is decidedly not available at this domain".)

4. The initiating entity chooses at least one of the returned FQDNs to resolve (following the rules in [DNS-SRV]), which it does by performing DNS "A" or "AAAA" lookups on the FDQN; this will result in an IPv4 or IPv6 address.

5. The initiating entity uses the IP address(es) from the successfully resolved FDQN (with the corresponding port number returned by the SRV lookup) as the connection address for the receiving entity.

6. If the initiating entity fails to connect using that IP address but the "A" or "AAAA" lookups returned more than one IP address, then the initiating entity uses the next resolved IP address for that FDQN as the connection address.

7. If the initiating entity fails to connect using all resolved IP addresses for a given FDQN, then it repeats the process of resolution and connection for the next FQDN returned by the SRV lookup based on the priority and weight as defined in [DNS-SRV].

8. If the initiating entity receives a response to its SRV query but it is not able to establish an XMPP connection using the data received in the response, it SHOULD NOT attempt the fallback process described in the next section (this helps to prevent a state mismatch between inbound and outbound connections).

9. If the initiating entity does not receive a response to its SRV query, it SHOULD attempt the fallback process described in the next section.

3.2.2. Fallback Processes

The fallback process SHOULD be a normal "A" or "AAAA" address record resolution to determine the IPv4 or IPv6 address of the origin domain, where the port used is the "xmpp-client" port of 5222 for client-to-server connections or the "xmpp-server" port of 5269 for server-to-server connections (these are the default ports as registered with the IANA as described under Section 14.7).

If connections via TCP are unsuccessful, the initiating entity might attempt to find and use alternative connection methods such as the HTTP binding (see [XEP-0124] and [XEP-0206]), which might be discovered using [DNS-TXT] records as described in [XEP-0156].

3.2.3. When Not to Use SRV

If the initiating entity has been explicitly configured to associate a particular FQDN (and potentially port) with the origin domain of the receiving entity (say, to "hardcode" an association from an origin domain of example.net to a configured FQDN of apps.example.com), the initiating entity is encouraged to use the configured name instead of performing the preferred SRV resolution process on the origin domain.

3.2.4. Use of SRV Records with Add-On Services

Many XMPP servers are implemented in such a way that they can host add-on services (beyond those defined in this specification and [XMPP-IM]) at DNS domain names that typically are "subdomains" of the main XMPP service (e.g., conference.example.net for a [XEP-0045] service associated with the example.net XMPP service) or "subdomains" of the first-level domain of the underlying service (e.g., muc.example.com for a [XEP-0045] service associated with the im.example.com XMPP service). If an entity associated with a remote XMPP server wishes to communicate with such an add-on service, it would generate an appropriate XML stanza and the remote server would attempt to resolve the add-on service's DNS domain name via an SRV lookup on resource records such as "_xmpp-server._tcp.conference.example.net." or "_xmpp-server._tcp.muc.example.com.". Therefore, if the administrators of an XMPP service wish to enable entities associated with remote servers to access such add-on services, they need to advertise the appropriate "_xmpp-server" SRV records in addition to the "_xmpp-server" record for their main XMPP service. In case SRV records are not available, the fallback methods described under Section 3.2.2 can be used to resolve the DNS domain names of add-on services.

3.3. Reconnection

It can happen that an XMPP server goes offline unexpectedly while servicing TCP connections from connected clients and remote servers. Because the number of such connections can be quite large, the reconnection algorithm employed by entities that seek to reconnect can have a significant impact on software performance and network congestion. If an entity chooses to reconnect, it:

- SHOULD set the number of seconds that expire before reconnecting to an unpredictable number between 0 and 60 (this helps to ensure that not all entities attempt to reconnect at exactly the same number of seconds after being disconnected).

- SHOULD back off increasingly on the time between subsequent reconnection attempts (e.g., in accordance with "truncated binary exponential backoff" as described in [ETHERNET]) if the first reconnection attempt does not succeed.

It is RECOMMENDED to make use of TLS session resumption [TLS-RESUME] when reconnecting. A future version of this document, or a separate specification, might provide more detailed guidelines regarding methods for speeding the reconnection process.

3.4. Reliability

The use of long-lived TCP connections in XMPP implies that the sending of XML stanzas over XML streams can be unreliable, since the parties to a long-lived TCP connection might not discover a connectivity disruption in a timely manner. At the XMPP application layer, long connectivity disruptions can result in undelivered stanzas. Although the core XMPP technology defined in this specification does not contain features to overcome this lack of reliability, there exist XMPP extensions for doing so (e.g., [XEP-0198]).

4. XML Streams

4.1. Stream Fundamentals

Two fundamental concepts make possible the rapid, asynchronous exchange of relatively small payloads of structured information between XMPP 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 "stream header" (i.e., an 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 complete TLS negotiation (Section 5) or SASL negotiation (Section 6)) or XML stanzas. The "initial stream" is negotiated from the initiating entity (typically a client or server) to the receiving entity (typically a server), and can be seen as corresponding to the initiating entity's "connection to" or "session with" the receiving entity. The initial stream enables unidirectional communication from the initiating entity to the receiving entity; in order to enable exchange of stanzas 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 the basic unit of meaning in XMPP. A stanza is a first-level element (at depth=1 of the stream) whose element name is "message", "presence", or "iq" and whose qualifying namespace is 'jabber:client' or 'jabber:server'. By contrast, a first-level element qualified by any other namespace is not an XML stanza (stream errors, stream features, TLS-related elements, SASL-related elements, etc.), nor is a

<message/>, <presence/>, or <iq/> element that is qualified by the 'jabber:client' or 'jabber:server' namespace but that occurs at a depth other than one (e.g., a <message/> element contained within an extension element (Section 8.4) for reporting purposes), nor is a <message/>, <presence/>, or <iq/> element that is qualified by a namespace other than 'jabber:client' or 'jabber:server'. An XML stanza typically contains one or more child elements (with accompanying attributes, elements, and XML character data) as necessary in order to convey the desired information, which MAY be qualified by any XML namespace (see [XML-NAMES] as well as Section 8.4 in this specification).

There are three kinds of stanzas: message, presence, and IQ (short for "Info/Query"). These stanza types provide three different communication primitives: a "push" mechanism for generalized messaging, a specialized "publish-subscribe" mechanism for broadcasting information about network availability, and a "request-response" mechanism for more structured exchanges of data (similar to [HTTP]). Further explanations are provided under Section 8.2.1, Section 8.2.2, and Section 8.2.3, respectively.

Consider the example of a client's connection to a server. The client initiates an XML stream by sending a stream header to the server, preferably preceded by an XML declaration specifying the XML version and the character encoding supported (see Section 11.5 and Section 11.6). Subject to local policies and service provisioning, the server then replies with a second XML stream back to the client, again preferably preceded by an XML declaration. Once the client has completed SASL negotiation (Section 6) and resource binding (Section 7), the client can send an unbounded number of XML stanzas over the stream. When the client desires to close the stream, it simply sends a closing </stream> tag to the server as further described under Section 4.4.

In essence, then, one XML stream functions as an envelope for the XML stanzas sent during a session and another XML stream functions as an envelope for the XML stanzas received during a session. We can represent this in a simplistic fashion as follows.

```
   +--------------------+--------------------+
   | INITIAL STREAM     |  RESPONSE STREAM   |
   +--------------------+--------------------+
   | <stream>           |                    |
   |--------------------|--------------------|
   |                    | <stream>           |
   |--------------------|--------------------|
   | <presence>         |                    |
   |   <show/>          |                    |
   | </presence>        |                    |
   |--------------------|--------------------|
   | <message to='foo'> |                    |
   |   <body/>          |                    |
   | </message>         |                    |
   |--------------------|--------------------|
   | <iq to='bar'       |                    |
   |     type='get'>    |                    |
   |   <query/>         |                    |
   | </iq>              |                    |
   |--------------------|--------------------|
   |                    | <iq from='bar'     |
   |                    |     type='result'> |
   |                    |   <query/>         |
   |                    | </iq>              |
   |--------------------|--------------------|
   | [ ... ]            |                    |
   |--------------------|--------------------|
   |                    | [ ... ]            |
   |--------------------|--------------------|
   | </stream>          |                    |
   |--------------------|--------------------|
   |                    | </stream>          |
   +--------------------+--------------------+
```

Figure 2: A Simplistic View of Two Streams

Those who are accustomed to thinking of XML in a document-centric manner might find the following analogies useful:

- o The two XML streams are like two "documents" (matching the "document" production from [XML]) that are built up through the accumulation of XML stanzas.

- o The root <stream/> element is like the "document entity" for each "document" (as described in Section 4.8 of [XML]).

- o The XML stanzas sent over the streams are like "fragments" of the "documents" (as described in [XML-FRAG]).

However, these descriptions are merely analogies, because XMPP does not deal in documents and fragments but in streams and stanzas.

The remainder of this section defines the following aspects of XML streams (along with related topics):

- How to open a stream (Section 4.2)
- The stream negotiation process (Section 4.3)
- How to close a stream (Section 4.4)
- The directionality of XML streams (Section 4.5)
- How to handle peers that are silent (Section 4.6)
- The XML attributes of a stream (Section 4.7)
- The XML namespaces of a stream (Section 4.8)
- Error handling related to XML streams (Section 4.9)

4.2. Opening a Stream

After connecting to the appropriate IP address and port of the receiving entity, the initiating entity opens a stream by sending a stream header (the "initial stream header") to the receiving entity.

```
I: <?xml version='1.0'?>
   <stream:stream
       from='juliet@im.example.com'
       to='im.example.com'
       version='1.0'
       xml:lang='en'
       xmlns='jabber:client'
       xmlns:stream='http://etherx.jabber.org/streams'>
```

The receiving entity then replies by sending a stream header of its own (the "response stream header") to the initiating entity.

```
   R: <?xml version='1.0'?>
      <stream:stream
          from='im.example.com'
          id='++TR84Sm6A3hnt3Q065SnAbbk3Y='
          to='juliet@im.example.com'
          version='1.0'
          xml:lang='en'
          xmlns='jabber:client'
          xmlns:stream='http://etherx.jabber.org/streams'>
```

The entities can then proceed with the remainder of the stream negotiation process.

4.3. Stream Negotiation

4.3.1. Basic Concepts

Because the receiving entity for a stream acts as a gatekeeper to the domains it services, it imposes certain conditions for connecting as a client or as a peer server. At a minimum, the initiating entity needs to authenticate with the receiving entity before it is allowed to send stanzas to the receiving entity (for client-to-server streams this means using SASL as described under Section 6). However, the receiving entity can consider conditions other than authentication to be mandatory-to-negotiate, such as encryption using TLS as described under Section 5. The receiving entity informs the initiating entity about such conditions by communicating "stream features": the set of particular protocol interactions that the initiating entity needs to complete before the receiving entity will accept XML stanzas from the initiating entity, as well as any protocol interactions that are voluntary-to-negotiate but that might improve the handling of an XML stream (e.g., establishment of application-layer compression as described in [XEP-0138]).

The existence of conditions for connecting implies that streams need to be negotiated. The order of layers (TCP, then TLS, then SASL, then XMPP as described under Section 13.3) implies that stream negotiation is a multi-stage process. Further structure is imposed by two factors: (1) a given stream feature might be offered only to certain entities or only after certain other features have been negotiated (e.g., resource binding is offered only after SASL authentication), and (2) stream features can be either mandatory-to-negotiate or voluntary-to-negotiate. Finally, for security reasons the parties to a stream need to discard knowledge that they gained during the negotiation process after successfully completing the protocol interactions defined for certain features (e.g., TLS in all cases and SASL in the case when a security layer might be established, as defined in the specification for the relevant SASL mechanism). This is done by flushing the old stream context and exchanging new stream headers over the existing TCP connection.

4.3.2. Stream Features Format

If the initiating entity includes in the initial stream header the 'version' attribute set to a value of at least "1.0" (see Section 4.7.5), after sending the response stream header the receiving entity MUST send a <features/> child element (typically prefixed by the stream namespace prefix as described under Section 4.8.5) to the initiating entity in order to announce any conditions for continuation of the stream negotiation process. Each condition takes the form of a child element of the <features/> element, qualified by a namespace that is different from the stream namespace and the content namespace. The <features/> element can contain one child, contain multiple children, or be empty.

Implementation Note: The order of child elements contained in any given <features/> element is not significant.

If a particular stream feature is or can be mandatory-to-negotiate, the definition of that feature needs to do one of the following:

1. Declare that the feature is always mandatory-to-negotiate (e.g., this is true of resource binding for XMPP clients); or

2. Specify a way for the receiving entity to flag the feature as mandatory-to-negotiate for this interaction (e.g., for STARTTLS, this is done by including an empty <required/> element in the advertisement for that stream feature, but that is not a generic format for all stream features); it is RECOMMENDED that stream feature definitions for new mandatory-to-negotiate features do so by including an empty <required/> element as is done for STARTTLS.

Informational Note: Because there is no generic format for indicating that a feature is mandatory-to-negotiate, it is possible that a feature that is not understood by the initiating entity might be considered mandatory-to-negotiate by the receiving entity, resulting in failure of the stream negotiation process. Although such an outcome would be undesirable, the working group deemed it rare enough that a generic format was not needed.

For security reasons, certain stream features necessitate the initiating entity to send a new initial stream header upon successful negotiation of the feature (e.g., TLS in all cases and SASL in the case when a security layer might be established). If this is true of a given stream feature, the definition of that feature needs to specify that a stream restart is expected after negotiation of the feature.

A <features/> element that contains at least one mandatory-to-negotiate feature indicates that the stream negotiation is not complete and that the initiating entity MUST negotiate further features.

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

A <features/> element MAY contain more than one mandatory-to-negotiate feature. This means that the initiating entity can choose among the mandatory-to-negotiate features at this stage of the stream negotiation process. As an example, perhaps a future technology will perform roughly the same function as TLS, so the receiving entity might advertise support for both TLS and the future technology at the same stage of the stream negotiation process. However, this applies only at a given stage of the stream negotiation process and does not apply to features that are mandatory-to-negotiate at different stages (e.g., the receiving entity would not advertise both STARTTLS and SASL as mandatory-to-negotiate, or both SASL and resource binding as mandatory-to-negotiate, because TLS would need to be negotiated before SASL and because SASL would need to be negotiated before resource binding).

A <features/> element that contains both mandatory-to-negotiate and voluntary-to-negotiate features indicates that the negotiation is not complete but that the initiating entity MAY complete the voluntary-to-negotiate feature(s) before it attempts to negotiate the mandatory-to-negotiate feature(s).

```
R: <stream:features>
     <bind xmlns='urn:ietf:params:xml:ns:xmpp-bind'/>
     <compression xmlns='http://jabber.org/features/compress'>
       <method>zlib</method>
       <method>lzw</method>
     </compression>
   </stream:features>
```

A <features/> element that contains only voluntary-to-negotiate features indicates that the stream negotiation is complete and that the initiating entity is cleared to send XML stanzas, but that the initiating entity MAY negotiate further features if desired.

```
R: <stream:features>
     <compression xmlns='http://jabber.org/features/compress'>
       <method>zlib</method>
       <method>lzw</method>
     </compression>
   </stream:features>
```

An empty <features/> element indicates that the stream negotiation is complete and that the initiating entity is cleared to send XML stanzas.

```
R: <stream:features/>
```

4.3.3. Restarts

On successful negotiation of a feature that necessitates a stream restart, both parties MUST consider the previous stream to be replaced but MUST NOT send a closing </stream> tag and MUST NOT terminate the underlying TCP connection; instead, the parties MUST reuse the existing connection, which might be in a new state (e.g., encrypted as a result of TLS negotiation). The initiating entity then MUST send a new initial stream header, which SHOULD be preceded by an XML declaration as described under Section 11.5. When the receiving entity receives the new initial stream header, it MUST generate a new stream ID (instead of reusing the old stream ID) before sending a new response stream header (which SHOULD be preceded by an XML declaration as described under Section 11.5).

4.3.4. Resending Features

The receiving entity MUST send an updated list of stream features to the initiating entity after a stream restart. The list of updated features MAY be empty if there are no further features to be advertised or MAY include any combination of features.

4.3.5. Completion of Stream Negotiation

The receiving entity indicates completion of the stream negotiation process by sending to the initiating entity either an empty <features/> element or a <features/> element that contains only voluntary-to-negotiate features. After doing so, the receiving entity MAY send an empty <features/> element (e.g., after negotiation of such voluntary-to-negotiate features) but MUST NOT send additional stream features to the initiating entity (if the receiving entity has new features to offer, preferably limited to mandatory-to-negotiate or security-critical features, it can simply close the stream with a <reset/> stream error (Section 4.9.3.16) and then advertise the new features when the initiating entity reconnects, preferably closing existing streams in a staggered way so that not all of the initiating entities reconnect at once). Once stream negotiation is complete, the initiating entity is cleared to send XML stanzas over the stream for as long as the stream is maintained by both parties.

Informational Note: Resource binding as specified under Section 7 is an historical exception to the foregoing rule, since it is mandatory-to-negotiate for clients but uses XML stanzas for negotiation purposes.

The initiating entity MUST NOT attempt to send XML stanzas (Section 8) to entities other than itself (i.e., the client's connected resource or any other authenticated resource of the client's account) or the server to which it is connected until stream negotiation has been completed. Even if the initiating entity does attempt to do so, the receiving entity MUST NOT accept such stanzas and MUST close the stream with a <not-authorized/> stream error (Section 4.9.3.12). This rule applies to XML stanzas only (i.e., <message/>, <presence/>, and <iq/> elements qualified by the content namespace) and not to XML elements used for stream negotiation (e.g., elements used to complete TLS negotiation (Section 5) or SASL negotiation (Section 6)).

4.3.6. Determination of Addresses

After the parties to an XML stream have completed the appropriate aspects of stream negotiation, the receiving entity for a stream MUST determine the initiating entity's JID.

For client-to-server communication, both SASL negotiation (Section 6) and resource binding (Section 7) MUST be completed before the server can determine the client's address. The client's bare JID (<localpart@domainpart>) MUST be the authorization identity (as defined by [SASL]), either (1) as directly communicated by the client during SASL negotiation (Section 6) or (2) as derived by the server from the authentication identity if no authorization identity was specified during SASL negotiation. The resourcepart of the full JID (<localpart@domainpart/resourcepart>) MUST be the resource negotiated by the client and server during resource binding (Section 7). A client MUST NOT attempt to guess at its JID but instead MUST consider its JID to be whatever the server returns to it during resource binding. The server MUST ensure that the resulting JID (including localpart, domainpart, resourcepart, and separator characters) conforms to the canonical format for XMPP addresses defined in [XMPP-ADDR]; to meet this restriction, the server MAY replace the JID sent by the client with the canonicalized JID as determined by the server and communicate that JID to the client during resource binding.

For server-to-server communication, the initiating server's bare JID (<domainpart>) MUST be the authorization identity (as defined by [SASL]), either (1) as directly communicated by the initiating server during SASL negotiation (Section 6) or (2) as derived by the receiving server from the authentication identity if no authorization identity was specified during SASL negotiation. In the absence of SASL negotiation, the receiving server MAY consider the authorization identity to be an identity negotiated within the relevant verification protocol (e.g., the 'from' attribute of the <result/> element in Server Dialback [XEP-0220]).

Security Warning: Because it is possible for a third party to tamper with information that is sent over the stream before a security layer such as TLS is successfully negotiated, it is advisable for the receiving server to treat any such unprotected information with caution; this applies especially to the 'from' and 'to' addresses on the first initial stream header sent by the initiating entity.

4.3.7. Flow Chart

We summarize the foregoing rules in the following non-normative flow chart for the stream negotiation process, presented from the perspective of the initiating entity.

```
                   +---------------------+
                   | open TCP connection |
                   +---------------------+
                              |
                              v
                       +---------------+
                       | send initial  |<-------------------------+
                       | stream header |                          ^
                       +---------------+                          |
                              |                                   |
                              v                                   |
                      +------------------+                        |
                      | receive response |                        |
                      | stream header    |                        |
                      +------------------+                        |
                              |                                   |
                              v                                   |
                       +----------------+                         |
                       | receive stream |                         |
   +------------------>| features       |                         |
   ^   {OPTIONAL}      +----------------+                         |
   |                          |                                   |
   |                          v                                   |
   |       +<-----------------+                                   |
   |       |                                                      |
   |    {empty?} ----> {all voluntary?} ----> {some mandatory?}   |
   |       |      no          |          no         |             |
   |       | yes              | yes                 | yes         |
   |       |                  v                     v             |
   |       |           +---------------+    +----------------+    |
   |       |           | MAY negotiate |    | MUST negotiate |    |
   |       |           | any or none   |    | one feature    |    |
   |       |           +---------------+    +----------------+    |
   |       v                  |                     |             |
   |   +---------+            v                     |             |
   |   |  DONE   |<----- {negotiate?}               |             |
   |   +---------+   no       |                     |             |
   |                     yes  |                     |             |
   |                          v                     v             |
   |                          +--------->+<---------+             |
   |                                     |                        |
   |                                     v                        |
   +<-------------------------- {restart mandatory?} ------------>+
                  no                                     yes
```

Figure 3: Stream Negotiation Flow Chart

4.4. Closing a Stream

An XML stream from one entity to another can be closed at any time, either because a specific stream error (Section 4.9) has occurred or in the absence of an error (e.g., when a client simply ends its session).

A stream is closed by sending a closing </stream> tag.

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

If the parties are using either two streams over a single TCP connection or two streams over two TCP connections, the entity that sends the closing stream tag MUST behave as follows:

1. Wait for the other party to also close its outbound stream before terminating the underlying TCP connection(s); this gives the other party an opportunity to finish transmitting any outbound data to the closing entity before the termination of the TCP connection(s).

2. Refrain from sending any further data over its outbound stream to the other entity, but continue to process data received from the other entity (and, if necessary, process such data).

3. Consider both streams to be void if the other party does not send its closing stream tag within a reasonable amount of time (where the definition of "reasonable" is a matter of implementation or deployment).

4. After receiving a reciprocal closing stream tag from the other party or waiting a reasonable amount of time with no response, terminate the underlying TCP connection(s).

> Security Warning: In accordance with Section 7.2.1 of [TLS], to help prevent a truncation attack the party that is closing the stream MUST send a TLS close_notify alert and MUST receive a responding close_notify alert from the other party before terminating the underlying TCP connection(s).

If the parties are using multiple streams over multiple TCP connections, there is no defined pairing of streams and therefore the behavior is a matter for implementation.

4.5. Directionality

An XML stream is always unidirectional, by which is meant that XML stanzas can be sent in only one direction over the stream (either from the initiating entity to the receiving entity or from the receiving entity to the initiating entity).

Depending on the type of session that has been negotiated and the nature of the entities involved, the entities might use:

- Two streams over a single TCP connection, where the security context negotiated for the first stream is applied to the second stream. This is typical for client-to-server sessions, and a server MUST allow a client to use the same TCP connection for both streams.

- Two streams over two TCP connections, where each stream is separately secured. In this approach, one TCP connection is used for the stream in which stanzas are sent from the initiating entity to the receiving entity, and the other TCP connection is used for the stream in which stanzas are sent from the receiving entity to the initiating entity. This is typical for server-to-server sessions.

- Multiple streams over two or more TCP connections, where each stream is separately secured. This approach is sometimes used for server-to-server communication between two large XMPP service providers; however, this can make it difficult to maintain coherence of data received over multiple streams in situations described under Section 10.1, which is why a server MAY close the stream with a <conflict/> stream error (Section 4.9.3.3) if a remote server attempts to negotiate more than one stream (as described under Section 4.9.3.3).

This concept of directionality applies only to stanzas and explicitly does not apply to first-level children of the stream root that are used to bootstrap or manage the stream (e.g., first-level elements used for TLS negotiation, SASL negotiation, Server Dialback [XEP-0220], and Stream Management [XEP-0198]).

The foregoing considerations imply that while completing STARTTLS negotiation (Section 5) and SASL negotiation (Section 6) two servers would use one TCP connection, but after the stream negotiation process is done that original TCP connection would be used only for the initiating server to send XML stanzas to the receiving server. In order for the receiving server to send XML stanzas to the initiating server, the receiving server would need to reverse the roles and negotiate an XML stream from the receiving server to the initiating server over a separate TCP connection. This separate TCP connection is then secured using a new round of TLS and/or SASL negotiation.

Implementation Note: For historical reasons, a server-to-server session always uses two TCP connections. While that approach remains the standard behavior described in this document, extensions such as [XEP-0288] enable servers to negotiate the use of a single TCP connection for bidirectional stanza exchange.

Informational Note: Although XMPP developers sometimes apply the terms "unidirectional" and "bidirectional" to the underlying TCP connection (e.g., calling the TCP connection for a client-to-server session "bidirectional" and the TCP connection for a server-to-server session "unidirectional"), strictly speaking a stream is always unidirectional (because the initiating entity and receiving entity always have a minimum of two streams, one in each direction) and a TCP connection is always bidirectional (because TCP traffic can be sent in both directions). Directionality applies to the application-layer traffic sent over the TCP connection, not to the transport-layer traffic sent over the TCP connection itself.

4.6. Handling of Silent Peers

When an entity that is a party to a stream has not received any XMPP traffic from its stream peer for some period of time, the peer might appear to be silent. There are several reasons why this might happen:

1. The underlying TCP connection is dead.

2. The XML stream is broken despite the fact that the underlying TCP connection is alive.

3. The peer is idle and simply has not sent any XMPP traffic over its XML stream to the entity.

These three conditions are best handled separately, as described in the following sections.

Implementation Note: For the purpose of handling silent peers, we treat a two unidirectional TCP connections as conceptually equivalent to a single bidirectional TCP connection (see Section 4.5); however, implementers need to be aware that, in the case of two unidirectional TCP connections, responses to traffic at the XMPP application layer will come back from the peer on the second TCP connection. In addition, the use of multiple streams in each direction (which is a somewhat frequent deployment choice for server-to-server connectivity among large XMPP service providers) further complicates application-level checking of XMPP streams and their underlying TCP connections, because there is no necessary correlation between any given initial stream and any given response stream.

4.6.1. Dead Connection

If the underlying TCP connection is dead, stream-level checks (e.g., [XEP-0199] and [XEP-0198]) are ineffective. Therefore, it is unnecessary to close the stream with or without an error, and it is appropriate instead to simply terminate the TCP connection.

One common method for checking the TCP connection is to send a space character (U+0020) between XML stanzas, which is allowed for XML streams as described under Section 11.7; the sending of such a space character is properly called a "whitespace keepalive" (the term "whitespace ping" is often used, despite the fact that it is not a ping since no "pong" is possible). However, this is not allowed during TLS negotiation or SASL negotiation, as described under Section 5.3.3 and Section 6.3.5.

4.6.2. Broken Stream

Even if the underlying TCP connection is alive, the peer might never respond to XMPP traffic that the entity sends, whether normal stanzas or specialized stream-checking traffic such as the application-level pings defined in [XEP-0199] or the more comprehensive Stream Management protocol defined in [XEP-0198]. In this case, it is appropriate for the entity to close a broken stream with a <connection-timeout/> stream error (Section 4.9.3.4).

4.6.3. Idle Peer

Even if the underlying TCP connection is alive and the stream is not broken, the peer might have sent no stanzas for a certain period of time. In this case, the peer itself MAY close the stream (as described under Section 4.4) rather than leaving an unused stream open. If the idle peer does not close the stream, the other party MAY either close the stream using the handshake described under Section 4.4 or close the stream with a stream error (e.g., <resource-constraint/> (Section 4.9.3.17) if the entity has reached a limit on the number of open TCP connections or <policy-violation/> (Section 4.9.3.14) if the connection has exceeded a local timeout policy). However, consistent with the order of layers (specified under Section 13.3), the other party is advised to verify that the underlying TCP connection is alive and the stream is unbroken (as described above) before concluding that the peer is idle. Furthermore, it is preferable to be liberal in accepting idle peers, since experience has shown that doing so improves the reliability of communication over XMPP networks and that it is typically more efficient to maintain a stream between two servers than to aggressively time out such a stream.

4.6.4. Use of Checking Methods

Implementers are advised to support whichever stream-checking and connection-checking methods they deem appropriate, but to carefully weigh the network impact of such methods against the benefits of discovering broken streams and dead TCP connections in a timely manner. The length of time between the use of any particular check is very much a matter of local service policy and depends strongly on the network environment and usage scenarios of a given deployment and connection type. At the time of writing, it is RECOMMENDED that any such check be performed not more than once every 5 minutes and that, ideally, such checks will be initiated by clients rather than servers. Those who implement XMPP software and deploy XMPP services are encouraged to seek additional advice regarding appropriate timing of stream-checking and connection-checking methods, particularly when power-constrained devices are being used (e.g., in mobile environments).

4.7. Stream Attributes

The attributes of the root <stream/> element are defined in the following sections.

Security Warning: Until and unless the confidentiality and integrity of the stream are protected via TLS as described under Section 5 or an equivalent security layer (such as the SASL GSSAPI mechanism), the attributes provided in a stream header could be tampered with by an attacker.

Implementation Note: The attributes of the root <stream/> element are not prepended by a namespace prefix because, as explained in [XML-NAMES], "[d]efault namespace declarations do not apply directly to attribute names; the interpretation of unprefixed attributes is determined by the element on which they appear."

4.7.1. from

The 'from' attribute specifies an XMPP identity of the entity sending the stream element.

For initial stream headers in client-to-server communication, the 'from' attribute is the XMPP identity of the principal controlling the client, i.e., a JID of the form <localpart@domainpart>. The client might not know the XMPP identity, e.g., because the XMPP identity is assigned at a level other than the XMPP application layer (as in the Generic Security Service Application Program Interface [GSS-API]) or is derived by the server from information provided by the client (as in some deployments of end-user certificates with the SASL EXTERNAL mechanism). Furthermore, if the client considers the XMPP identity to be private information then it is advised not to include a 'from' attribute before the confidentiality and integrity of the stream are protected via TLS or an equivalent security layer. However, if the client knows the XMPP identity then it SHOULD include the 'from' attribute after the confidentiality and integrity of the stream are protected via TLS or an equivalent security layer.

```
I: <?xml version='1.0'?>
   <stream:stream
       from='juliet@im.example.com'
       to='im.example.com'
       version='1.0'
       xml:lang='en'
       xmlns='jabber:client'
       xmlns:stream='http://etherx.jabber.org/streams'>
```

For initial stream headers in server-to-server communication, the 'from' attribute is one of the configured FQDNs of the server, i.e., a JID of the form <domainpart>. The initiating server might have more than one XMPP identity, e.g., in the case of a server that provides virtual hosting, so it will need to choose an identity that is associated with this output stream (e.g., based on the 'to' attribute of the stanza that triggered the stream negotiation attempt). Because a server is a "public entity" on the XMPP network, it MUST include the 'from' attribute after the confidentiality and integrity of the stream are protected via TLS or an equivalent security layer.

```
I: <?xml version='1.0'?>
   <stream:stream
       from='example.net'
       to='im.example.com'
       version='1.0'
       xml:lang='en'
       xmlns='jabber:server'
       xmlns:stream='http://etherx.jabber.org/streams'>
```

For response stream headers in both client-to-server and server-to-server communication, the receiving entity MUST include the 'from' attribute and MUST set its value to one of the receiving entity's FQDNs (which MAY be an FQDN other than that specified in the 'to' attribute of the initial stream header, as described under Section 4.9.1.3 and Section 4.9.3.6).

```
R: <?xml version='1.0'?>
   <stream:stream
       from='im.example.com'
       id='++TR84Sm6A3hnt3Q065SnAbbk3Y='
       to='juliet@im.example.com'
       version='1.0'
       xml:lang='en'
       xmlns='jabber:client'
       xmlns:stream='http://etherx.jabber.org/streams'>
```

Whether or not the 'from' attribute is included, each entity MUST verify the identity of the other entity before exchanging XML stanzas with it, as described under Section 13.5.

> Interoperability Note: It is possible that implementations based on [RFC3920] will not include the 'from' address on any stream headers (even ones whose confidentiality and integrity are protected); an entity SHOULD be liberal in accepting such stream headers.

4.7.2. to

For initial stream headers in both client-to-server and server-to-server communication, the initiating entity MUST include the 'to' attribute and MUST set its value to a domainpart that the initiating entity knows or expects the receiving entity to service. (The same information can be provided in other ways, such as a Server Name Indication during TLS negotiation as described in [TLS-EXT].)

```
I: <?xml version='1.0'?>
   <stream:stream
       from='juliet@im.example.com'
       to='im.example.com'
       version='1.0'
       xml:lang='en'
       xmlns='jabber:client'
       xmlns:stream='http://etherx.jabber.org/streams'>
```

For response stream headers in client-to-server communication, if the client included a 'from' attribute in the initial stream header then the server MUST include a 'to' attribute in the response stream header and MUST set its value to the bare JID specified in the 'from' attribute of the initial stream header. If the client did not include a 'from' attribute in the initial stream header then the server MUST NOT include a 'to' attribute in the response stream header.

```
R: <?xml version='1.0'?>
   <stream:stream
       from='im.example.com'
       id='++TR84Sm6A3hnt3Q065SnAbbk3Y='
       to='juliet@im.example.com'
       version='1.0'
       xml:lang='en'
       xmlns='jabber:client'
       xmlns:stream='http://etherx.jabber.org/streams'>
```

For response stream headers in server-to-server communication, the receiving entity MUST include a 'to' attribute in the response stream header and MUST set its value to the domainpart specified in the 'from' attribute of the initial stream header.

```
R: <?xml version='1.0'?>
   <stream:stream
       from='im.example.com'
       id='g4qSvGvBxJ+xeAd7QKezOQJFFlw='
       to='example.net'
       version='1.0'
       xml:lang='en'
       xmlns='jabber:server'
       xmlns:stream='http://etherx.jabber.org/streams'>
```

Whether or not the 'to' attribute is included, each entity MUST verify the identity of the other entity before exchanging XML stanzas with it, as described under Section 13.5.

> Interoperability Note: It is possible that implementations based on [RFC3920] will not include the 'to' address on stream headers; an entity SHOULD be liberal in accepting such stream headers.

4.7.3. id

The 'id' attribute specifies a unique identifier for the stream, called a "stream ID". The stream ID MUST be generated by the receiving entity when it sends a response stream header and MUST BE unique within the receiving application (normally a server).

Security Warning: The stream ID MUST be both unpredictable and non-repeating because it can be security-critical when reused by an authentication mechanisms, as is the case for Server Dialback [XEP-0220] and the "XMPP 0.9" authentication mechanism used before RFC 3920 defined the use of SASL in XMPP; for recommendations regarding randomness for security purposes, see [RANDOM].

For initial stream headers, the initiating entity MUST NOT include the 'id' attribute; however, if the 'id' attribute is included, the receiving entity MUST ignore it.

For response stream headers, the receiving entity MUST include the 'id' attribute.

```
R: <?xml version='1.0'?>
   <stream:stream
       from='im.example.com'
       id='++TR84Sm6A3hnt3Q065SnAbbk3Y='
       to='juliet@im.example.com'
       version='1.0'
       xml:lang='en'
       xmlns='jabber:client'
       xmlns:stream='http://etherx.jabber.org/streams'>
```

Interoperability Note: In RFC 3920, the text regarding inclusion of the 'id' attribute was ambiguous, leading some implementations to leave the attribute off the response stream header.

4.7.4. xml:lang

The 'xml:lang' attribute specifies an entity's preferred or default language for any human-readable XML character data to be sent over the stream (an XML stanza can also possess an 'xml:lang' attribute, as discussed under Section 8.1.5). The syntax of this attribute is defined in Section 2.12 of [XML]; in particular, the value of the 'xml:lang' attribute MUST conform to the NMTOKEN datatype (as defined in Section 2.3 of [XML]) and MUST conform to the language identifier format defined in [LANGTAGS].

For initial stream headers, the initiating entity SHOULD include the 'xml:lang' attribute.

```
   I: <?xml version='1.0'?>
      <stream:stream
          from='juliet@im.example.com'
          to='im.example.com'
          version='1.0'
          xml:lang='en'
          xmlns='jabber:client'
          xmlns:stream='http://etherx.jabber.org/streams'>
```

For response stream headers, the receiving entity MUST include the 'xml:lang' attribute. The following rules apply:

o If the initiating entity included an 'xml:lang' attribute in its initial stream header and the receiving entity supports that language in the human-readable XML character data that it generates and sends to the initiating entity (e.g., in the <text/> element for stream and stanza errors), the value of the 'xml:lang' attribute MUST be the identifier for the initiating entity's preferred language (e.g., "de-CH").

o If the receiving entity supports a language that matches the initiating entity's preferred language according to the "lookup scheme" specified in Section 3.4 of [LANGMATCH] (e.g., "de" instead of "de-CH"), then the value of the 'xml:lang' attribute SHOULD be the identifier for the matching language.

o If the receiving entity does not support the initiating entity's preferred language or a matching language according to the lookup scheme (or if the initiating entity did not include the 'xml:lang' attribute in its initial stream header), then the value of the 'xml:lang' attribute MUST be the identifier for the default language of the receiving entity (e.g., "en").

```
   R: <?xml version='1.0'?>
      <stream:stream
          from='im.example.com'
          id='++TR84Sm6A3hnt3Q065SnAbbk3Y='
          to='juliet@im.example.com'
          version='1.0'
          xml:lang='en'
          xmlns='jabber:client'
          xmlns:stream='http://etherx.jabber.org/streams'>
```

If the initiating entity included the 'xml:lang' attribute in its initial stream header, the receiving entity SHOULD remember that value as the default xml:lang for all stanzas sent by the initiating entity over the current stream. As described under Section 8.1.5, the initiating entity MAY include the 'xml:lang' attribute in any XML stanzas it sends over the stream. If the initiating entity does not include the 'xml:lang' attribute in any such stanza, the receiving entity SHOULD add the 'xml:lang' attribute to the stanza when routing it to a remote server or delivering it to a connected client, where the value of the attribute MUST be the identifier for the language preferred by the initiating entity (even if the receiving entity does not support that language for human-readable XML character data it generates and sends to the initiating entity, such as in stream or stanza errors). If the initiating entity includes the 'xml:lang' attribute in any such stanza, the receiving entity MUST NOT modify or delete it when routing it to a remote server or delivering it to a connected client.

4.7.5. version

The inclusion of the version attribute set to a value of at least "1.0" signals support for the stream-related protocols defined in this specification, including TLS negotiation (Section 5), SASL negotiation (Section 6), stream features (Section 4.3.2), and stream errors (Section 4.9).

The version of XMPP specified in this specification is "1.0"; in particular, XMPP 1.0 encapsulates the stream-related protocols as well as the basic semantics of the three defined XML stanza types (<message/>, <presence/>, and <iq/> as described under Sections 8.2.1, 8.2.2, and 8.2.3, respectively).

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 will be incremented only if the stream and stanza formats or obligatory 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 defined in the older specification.

The minor version number will be incremented only if significant new capabilities have been added to the core protocol (e.g., a newly defined value of the 'type' attribute for message, presence, or IQ stanzas). The minor version number MUST be ignored by an entity with a smaller minor version number, but MAY be used for informational purposes by the entity with the larger minor version number (e.g., 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:

1. The initiating entity MUST set the value of the 'version' attribute in 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 in 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, the initiating entity SHOULD close the stream with an <unsupported-version/> stream error (Section 4.9.3.25).

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 "0.9" and SHOULD NOT include a 'version' attribute in the response stream header.

4.7.6. Summary of Stream Attributes

The following table summarizes the attributes of the root <stream/> element.

	initiating to receiving	receiving to initiating
to	JID of receiver	JID of initiator
from	JID of initiator	JID of receiver
id	ignored	stream identifier
xml:lang	default language	default language
version	XMPP 1.0+ supported	XMPP 1.0+ supported

Figure 4: Stream Attributes

4.8. XML Namespaces

Readers are referred to the specification of XML namespaces [XML-NAMES] for a full understanding of the concepts used in this section, especially the concept of a "default namespace" as provided in Section 3 and Section 6.2 of that specification.

4.8.1. Stream Namespace

The root <stream/> element ("stream header") MUST be qualified by the namespace 'http://etherx.jabber.org/streams' (the "stream namespace"). If this rule is violated, the entity that receives the offending stream header MUST close the stream with a stream error, which SHOULD be <invalid-namespace/> (Section 4.9.3.10), although some existing implementations send <bad-format/> (Section 4.9.3.1) instead.

4.8.2. Content Namespace

An entity MAY declare a "content namespace" as the default namespace for data sent over the stream (i.e., data other than elements qualified by the stream namespace). If so, (1) the content namespace MUST be other than the stream namespace, and (2) the content namespace MUST be the same for the initial stream and the response stream so that both streams are qualified consistently. The content namespace applies to all first-level child elements sent over the stream unless explicitly qualified by another namespace (i.e., the content namespace is the default namespace).

Alternatively (i.e., instead of declaring the content namespace as the default namespace), an entity MAY explicitly qualify the namespace for each first-level child element of the stream, using so-called "prefix-free canonicalization". These two styles are shown in the following examples.

When a content namespace is declared as the default namespace, in rough outline a stream will look something like the following.

```
<stream:stream
    from='juliet@im.example.com'
    to='im.example.com'
    version='1.0'
    xml:lang='en'
    xmlns='jabber:client'
    xmlns:stream='http://etherx.jabber.org/streams'>
  <message>
    <body>foo</body>
  </message>
</stream:stream>
```

When a content namespace is not declared as the default namespace and so-called "prefix-free canonicalization" is used instead, in rough outline a stream will look something like the following.

```
<stream
    from='juliet@im.example.com'
    to='im.example.com'
    version='1.0'
    xml:lang='en'
    xmlns='http://etherx.jabber.org/streams'>
  <message xmlns='jabber:client'>
    <body>foo</body>
  </message>
</stream>
```

Traditionally, most XMPP implementations have used the content-namespace-as-default-namespace style rather than the prefix-free canonicalization style for stream headers; however, both styles are acceptable since they are semantically equivalent.

4.8.3. XMPP Content Namespaces

XMPP as defined in this specification uses two content namespaces: 'jabber:client' and 'jabber:server'. These namespaces are nearly identical but are used in different contexts (client-to-server communication for 'jabber:client' and server-to-server communication for 'jabber:server'). The only difference between the two is that the 'to' and 'from' attributes are OPTIONAL on stanzas sent over XML streams qualified by the 'jabber:client' namespace, whereas they are REQUIRED on stanzas sent over XML streams qualified by the 'jabber:server' namespace. Support for these content namespaces implies support for the common attributes (Section 8.1) and basic semantics (Section 8.2) of all three core stanza types (message, presence, and IQ).

An implementation MAY support content namespaces other than 'jabber:client' or 'jabber:server'. However, because such namespaces would define applications other than XMPP, they are to be defined in separate specifications.

An implementation MAY refuse to support any other content namespaces as default namespaces. If an entity receives a first-level child element qualified by a content namespace it does not support, it MUST close the stream with an <invalid-namespace/> stream error (Section 4.9.3.10).

Client implementations MUST support the 'jabber:client' content namespace as a default namespace. The 'jabber:server' content namespace is out of scope for an XMPP client, and a client MUST NOT send stanzas qualified by the 'jabber:server' namespace.

Server implementations MUST support as default content namespaces both the 'jabber:client' namespace (when the stream is used for communication between a client and a server) and the 'jabber:server' namespace (when the stream is used for communication between two servers). When communicating with a connected client, a server MUST NOT send stanzas qualified by the 'jabber:server' namespace; when communicating with a peer server, a server MUST NOT send stanzas qualified by the 'jabber:client' namespace.

> Implementation Note: Because a client sends stanzas over a stream whose content namespace is 'jabber:client', if a server routes to a peer server a stanza it has received from a connected client then it needs to "re-scope" the stanza so that its content namespace is 'jabber:server'. Similarly, if a server delivers to a connected client a stanza it has received from a peer server then it needs to "re-scope" the stanza so that its content namespace is 'jabber:client'. This rule applies to XML stanzas as defined under Section 4.1 (i.e., a first-level <message/>, <presence/>, or <iq/> element qualified by the 'jabber:client' or 'jabber:server' namespace), and by namespace inheritance to all child elements of a stanza. However, the rule does not apply to elements qualified by namespaces other than 'jabber:client' and 'jabber:server' nor to any children of such elements (e.g., a <message/> element contained within an extension element

(Section 8.4) for reporting purposes). Although it is not forbidden for an entity to generate stanzas in which an extension element contains a child element qualified by the 'jabber:client' or 'jabber:server' namespace, existing implementations handle such stanzas inconsistently; therefore, implementers are advised to weigh the likely lack of interoperability against the possible utility of such stanzas. Finally, servers are advised to apply stanza re-scoping to other stream connection methods and alternative XMPP connection methods, such as those specified in [XEP-0124], [XEP-0206], [XEP-0114], and [XEP-0225].

4.8.4. Other Namespaces

Either party to a stream MAY send data qualified by namespaces other than the content namespace and the stream namespace. For example, this is how data related to TLS negotiation and SASL negotiation are exchanged, as well as XMPP extensions such as Stream Management [XEP-0198] and Server Dialback [XEP-0220].

Interoperability Note: For historical reasons, some server implementations expect a declaration of the 'jabber:server:dialback' namespace on server-to-server streams, as explained in [XEP-0220].

However, an XMPP server MUST NOT route or deliver data received over an input stream if that data is (a) qualified by another namespace and (b) addressed to an entity other than the server, unless the other party to the output stream over which the server would send the data has explicitly negotiated or advertised support for receiving arbitrary data from the server. This rule is included because XMPP is designed for the exchange of XML stanzas (not arbitrary XML data), and because allowing an entity to send arbitrary data to other entities could significantly increase the potential for exchanging malicious information. As an example of this rule, the server hosting the example.net domain would not route the following first-level XML element from <romeo@example.net> to <juliet@example.com>:

```
<ns1:foo xmlns:ns1='http://example.org/ns1'
         from='romeo@example.net/resource1'
         to='juliet@example.com'>
  <ns1:bar/>
</ns1:foo>
```

This rule also applies to first-level elements that look like stanzas but that are improperly namespaced and therefore really are not stanzas at all (see also Section 4.8.5), for example:

```
     <ns2:message xmlns:ns2='http://example.org/ns2'
                  from='romeo@example.net/resource1'
                  to='juliet@example.com'>
       <body>hi</body>
     </ns2:message>
```

Upon receiving arbitrary first-level XML elements over an input stream, a server MUST either ignore the data or close the stream with a stream error, which SHOULD be <unsupported-stanza-type/> (Section 4.9.3.24).

4.8.5. Namespace Declarations and Prefixes

Because the content namespace is other than the stream namespace, if a content namespace is declared as the default namespace then the following statements are true:

1. The stream header needs to contain a namespace declaration for both the content namespace and the stream namespace.

2. The stream namespace declaration needs to include a namespace prefix for the stream namespace.

   Interoperability Note: For historical reasons, an implementation MAY accept only the prefix 'stream' for the stream namespace (resulting in prefixed names such as <stream:stream> and <stream:features>); this specification retains that allowance from [RFC3920] for the purpose of backward compatibility. Implementations are advised that using a prefix other than 'stream' for the stream namespace might result in interoperability problems. If an entity receives a stream header with a stream namespace prefix it does not accept, it MUST close the stream with a stream error, which SHOULD be <bad-namespace-prefix/> (Section 4.9.3.2), although some existing implementations send <bad-format/> (Section 4.9.3.1) instead.

An implementation MUST NOT generate namespace prefixes for elements qualified by the content namespace (i.e., the default namespace for data sent over the stream) if the content namespace is 'jabber:client' or 'jabber:server'. For example, the following is illegal:

```
   <stream:stream
       from='juliet@im.example.com'
       to='im.example.com'
       version='1.0'
       xml:lang='en'
       xmlns='jabber:client'
       xmlns:stream='http://etherx.jabber.org/streams'>
```

```
     <foo:message xmlns:foo='jabber:client'>
       <foo:body>foo</foo:body>
     </foo:message>
```

An XMPP entity SHOULD NOT accept data that violates this rule (in particular, an XMPP server MUST NOT route or deliver such data to another entity without first correcting the error); instead it SHOULD either ignore the data or close the stream with a stream error, which SHOULD be <bad-namespace-prefix/> (Section 4.9.3.2).

Namespaces declared in a stream header MUST apply only to that stream (e.g., the 'jabber:server:dialback' namespace used in Server Dialback [XEP-0220]). In particular, because XML stanzas intended for routing or delivery over streams with other entities will lose the namespace context declared in the header of the stream in which those stanzas originated, namespaces for extended content within such stanzas MUST NOT be declared in that stream header (see also Section 8.4). If either party to a stream declares such namespaces, the other party to the stream SHOULD close the stream with an <invalid-namespace/> stream error (Section 4.9.3.10). In any case, an entity MUST ensure that such namespaces are properly declared (according to this section) when routing or delivering stanzas from an input stream to an output stream.

4.9. Stream Errors

The root stream element MAY contain an <error/> child element that is qualified by the stream namespace. The error child SHALL be sent by a compliant entity if it perceives that a stream-level error has occurred.

4.9.1. Rules

The following rules apply to stream-level errors.

4.9.1.1. Stream Errors Are Unrecoverable

Stream-level errors are unrecoverable. Therefore, if an error occurs at the level of the stream, the entity that detects the error MUST send an <error/> element with an appropriate child element specifying the error condition and then immediately close the stream as described under Section 4.4.

```
   C: <message><body>No closing tag!</message>

   S: <stream:error>
        <not-well-formed
            xmlns='urn:ietf:params:xml:ns:xmpp-streams'/>
      </stream:error>
      </stream:stream>
```

The entity that receives the stream error then SHALL close the stream as explained under Section 4.4.

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

4.9.1.2. Stream Errors Can Occur During Setup

If the error is triggered by the initial stream header, the receiving entity MUST still send the opening <stream> tag, include the <error/> element as a child of the stream element, and send the closing </stream> tag (preferably in the same TCP packet).

```
   C: <?xml version='1.0'?>
      <stream:stream
          from='juliet@im.example.com'
          to='im.example.com'
          version='1.0'
          xml:lang='en'
          xmlns='jabber:client'
          xmlns:stream='http://wrong.namespace.example.org/'>

   S: <?xml version='1.0'?>
      <stream:stream
          from='im.example.com'
          id='++TR84Sm6A3hnt3Q065SnAbbk3Y='
          to='juliet@im.example.com'
          version='1.0'
          xml:lang='en'
          xmlns='jabber:client'
          xmlns:stream='http://etherx.jabber.org/streams'>
      <stream:error>
        <invalid-namespace
            xmlns='urn:ietf:params:xml:ns:xmpp-streams'/>
      </stream:error>
      </stream:stream>
```

4.9.1.3. Stream Errors When the Host Is Unspecified or Unknown

If the initiating entity provides no 'to' attribute or provides an unknown host in the 'to' attribute and the error occurs during stream setup, the value of the 'from' attribute returned by the receiving entity in the stream header sent before closing the stream MUST be either an authoritative FQDN for the receiving entity or the empty string.

```
C: <?xml version='1.0'?>
   <stream:stream
       from='juliet@im.example.com'
       to='unknown.host.example.com'
       version='1.0'
       xml:lang='en'
       xmlns='jabber:client'
       xmlns:stream='http://etherx.jabber.org/streams'>

S: <?xml version='1.0'?>
   <stream:stream
       from='im.example.com'
       id='++TR84Sm6A3hnt3Q065SnAbbk3Y='
       to='juliet@im.example.com'
       version='1.0'
       xml:lang='en'
       xmlns='jabber:client'
       xmlns:stream='http://etherx.jabber.org/streams'>
   <stream:error>
     <host-unknown
         xmlns='urn:ietf:params:xml:ns:xmpp-streams'/>
   </stream:error>
   </stream:stream>
```

4.9.1.4. Where Stream Errors Are Sent

When two TCP connections are used between the initiating entity and the receiving entity (one in each direction) rather than using a single bidirectional connection, the following rules apply:

- Stream-level errors related to the initial stream are returned by the receiving entity on the response stream via the same TCP connection.

- Stanza errors triggered by outbound stanzas sent from the initiating entity over the initial stream via the same TCP connection are returned by the receiving entity on the response stream via the other ("return") TCP connection, since they are inbound stanzas from the perspective of the initiating entity.

4.9.2. Syntax

The syntax for stream errors is as follows, where XML data shown within the square brackets '[' and ']' is OPTIONAL.

```
<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 "defined-condition" MUST correspond to one of the stream error conditions defined under Section 4.9.3. However, because additional error conditions might be defined in the future, if an entity receives a stream error condition that it does not understand then it MUST treat the unknown condition as equivalent to <undefined-condition/> (Section 4.9.3.21). If the designers of an XMPP protocol extension or the developers of an XMPP implementation need to communicate a stream error condition that is not defined in this specification, they can do so by defining an application-specific error condition element qualified by an application-specific namespace.

The <error/> element:

- o MUST contain a child element corresponding to one of the defined stream error conditions (Section 4.9.3); this element MUST be qualified by the 'urn:ietf:params:xml:ns:xmpp-streams' namespace.

- o MAY contain a <text/> child element 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 (see Section 4.9.4).

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

4.9.3. Defined Stream Error Conditions

The following stream-level error conditions are defined.

4.9.3.1. bad-format

The entity has sent XML that cannot be processed.

(In the following example, the client sends an XMPP message that is not well-formed XML, which alternatively might trigger a <not-well-formed/> stream error (Section 4.9.3.13).)

```
C: <message>
     <body>No closing tag!
   </message>

S: <stream:error>
     <bad-format
         xmlns='urn:ietf:params:xml:ns:xmpp-streams'/>
   </stream:error>
   </stream:stream>
```

This error can be used instead of the more specific XML-related errors, such as <bad-namespace-prefix/>, <invalid-xml/>, <not-well-formed/>, <restricted-xml/>, and <unsupported-encoding/>. However, the more specific errors are RECOMMENDED.

4.9.3.2. bad-namespace-prefix

The entity has sent a namespace prefix that is unsupported, or has sent no namespace prefix on an element that needs such a prefix (see Section 11.2).

(In the following example, the client specifies a namespace prefix of "foobar" for the XML stream namespace.)

```
C: <?xml version='1.0'?>
   <foobar:stream
       from='juliet@im.example.com'
       to='im.example.com'
       version='1.0'
       xmlns='jabber:client'
       xmlns:foobar='http://etherx.jabber.org/streams'>
S: <?xml version='1.0'?>
   <stream:stream
       from='im.example.com'
       id='++TR84Sm6A3hnt3Q065SnAbbk3Y='
       to='juliet@im.example.com'
       version='1.0'
       xml:lang='en'
       xmlns='jabber:client'
       xmlns:stream='http://etherx.jabber.org/streams'>
   <stream:error>
     <bad-namespace-prefix
         xmlns='urn:ietf:params:xml:ns:xmpp-streams'/>
   </stream:error>
   </stream:stream>
```

4.9.3.3. conflict

The server either (1) is closing the existing stream for this entity because a new stream has been initiated that conflicts with the existing stream, or (2) is refusing a new stream for this entity because allowing the new stream would conflict with an existing stream (e.g., because the server allows only a certain number of connections from the same IP address or allows only one server-to-server stream for a given domain pair as a way of helping to ensure in-order processing as described under Section 10.1).

```
   C: <?xml version='1.0'?>
      <stream:stream
          from='juliet@im.example.com'
          to='im.example.com'
          version='1.0'
          xmlns='jabber:client'
          xmlns:stream='http://etherx.jabber.org/streams'>

   S: <?xml version='1.0'?>
      <stream:stream
          from='im.example.com'
          id='++TR84Sm6A3hnt3Q065SnAbbk3Y='
          to='juliet@im.example.com'
          version='1.0'
          xml:lang='en'
          xmlns='jabber:client'
          xmlns:stream='http://etherx.jabber.org/streams'>
      <stream:error>
        <conflict
            xmlns='urn:ietf:params:xml:ns:xmpp-streams'/>
      </stream:error>
      </stream:stream>
```

If a client receives a <conflict/> stream error (Section 4.9.3.3), during the resource binding aspect of its reconnection attempt it MUST NOT blindly request the resourcepart it used during the former session but instead MUST choose a different resourcepart; details are provided under Section 7.

4.9.3.4. connection-timeout

One party is closing the stream because it has reason to believe that the other party has permanently lost the ability to communicate over the stream. The lack of ability to communicate can be discovered using various methods, such as whitespace keepalives as specified under Section 4.4, XMPP-level pings as defined in [XEP-0199], and XMPP Stream Management as defined in [XEP-0198].

```
P: <stream:error>
     <connection-timeout
         xmlns='urn:ietf:params:xml:ns:xmpp-streams'/>
   </stream:error>
   </stream:stream>
```

Interoperability Note: RFC 3920 specified that the <connection-timeout/> stream error (Section 4.9.3.4) is to be used if the peer has not generated any traffic over the stream for some period of time. That behavior is no longer recommended; instead, the error SHOULD be used only if the connected client or peer server has not responded to data sent over the stream.

4.9.3.5. host-gone

The value of the 'to' attribute provided in the initial stream header corresponds to an FQDN that is no longer serviced by the receiving entity.

(In the following example, the peer specifies a 'to' address of "foo.im.example.com" when connecting to the "im.example.com" server, but the server no longer hosts a service at that address.)

```
P: <?xml version='1.0'?>
   <stream:stream
       from='example.net'
       to='foo.im.example.com'
       version='1.0'
       xmlns='jabber:server'
       xmlns:stream='http://etherx.jabber.org/streams'>
S: <?xml version='1.0'?>
   <stream:stream
       from='im.example.com'
       id='g4qSvGvBxJ+xeAd7QKezOQJFFlw='
       to='example.net'
       version='1.0'
       xml:lang='en'
       xmlns='jabber:server'
       xmlns:stream='http://etherx.jabber.org/streams'>
   <stream:error>
     <host-gone
         xmlns='urn:ietf:params:xml:ns:xmpp-streams'/>
   </stream:error>
   </stream:stream>
```

4.9.3.6. host-unknown

The value of the 'to' attribute provided in the initial stream header does not correspond to an FQDN that is serviced by the receiving entity.

(In the following example, the peer specifies a 'to' address of "example.org" when connecting to the "im.example.com" server, but the server knows nothing of that address.)

```
   P: <?xml version='1.0'?>
      <stream:stream
          from='example.net'
          to='example.org'
          version='1.0'
          xmlns='jabber:server'
          xmlns:stream='http://etherx.jabber.org/streams'>

   S: <?xml version='1.0'?>
      <stream:stream
          from='im.example.com'
          id='g4qSvGvBxJ+xeAd7QKezOQJFFlw='
          to='example.net'
          version='1.0'
          xml:lang='en'
          xmlns='jabber:server'
          xmlns:stream='http://etherx.jabber.org/streams'>
      <stream:error>
        <host-unknown
            xmlns='urn:ietf:params:xml:ns:xmpp-streams'/>
      </stream:error>
      </stream:stream>
```

4.9.3.7. improper-addressing

A stanza sent between two servers lacks a 'to' or 'from' attribute, the 'from' or 'to' attribute has no value, or the value violates the rules for XMPP addresses [XMPP-ADDR].

(In the following example, the peer sends a stanza without a 'to' address over a server-to-server stream.)

```
P: <message from='juliet@im.example.com'>
     <body>Wherefore art thou?</body>
   </message>

S: <stream:error>
     <improper-addressing
         xmlns='urn:ietf:params:xml:ns:xmpp-streams'/>
   </stream:error>
   </stream:stream>
```

4.9.3.8. internal-server-error

The server has experienced a misconfiguration or other internal error that prevents it from servicing the stream.

```
S: <stream:error>
     <internal-server-error
         xmlns='urn:ietf:params:xml:ns:xmpp-streams'/>
   </stream:error>
   </stream:stream>
```

4.9.3.9. invalid-from

The data provided in a 'from' attribute does not match an authorized JID or validated domain as negotiated (1) between two servers using SASL or Server Dialback, or (2) between a client and a server via SASL authentication and resource binding.

(In the following example, a peer that has authenticated only as "example.net" attempts to send a stanza from an address at "example.org".)

```
P: <message from='romeo@example.org' to='juliet@im.example.com'>
     <body>Neither, fair saint, if either thee dislike.</body>
   </message>
S: <stream:error>
     <invalid-from
         xmlns='urn:ietf:params:xml:ns:xmpp-streams'/>
   </stream:error>
   </stream:stream>
```

4.9.3.10. invalid-namespace

The stream namespace name is something other than "http://etherx.jabber.org/streams" (see Section 11.2) or the content namespace declared as the default namespace is not supported (e.g., something other than "jabber:client" or "jabber:server").

(In the following example, the client specifies a namespace of 'http://wrong.namespace.example.org/' for the stream.)

```
   C: <?xml version='1.0'?>
      <stream:stream
          from='juliet@im.example.com'
          to='im.example.com'
          version='1.0'
          xmlns='jabber:client'
          xmlns:stream='http://wrong.namespace.example.org/'>

   S: <?xml version='1.0'?>
      <stream:stream
          from='im.example.com'
          id='++TR84Sm6A3hnt3Q065SnAbbk3Y='
          to='juliet@im.example.com'
          version='1.0'
          xml:lang='en'
          xmlns='jabber:client'
          xmlns:stream='http://etherx.jabber.org/streams'>
      <stream:error>
        <invalid-namespace
            xmlns='urn:ietf:params:xml:ns:xmpp-streams'/>
      </stream:error>
      </stream:stream>
```

4.9.3.11. invalid-xml

The entity has sent invalid XML over the stream to a server that performs validation (see Section 11.4).

(In the following example, the peer attempts to send an IQ stanza of type "subscribe", but the XML schema defines no such value for the 'type' attribute.)

```
   P: <iq from='example.net'
          id='l3b1vs75'
          to='im.example.com'
          type='subscribe'>
        <ping xmlns='urn:xmpp:ping'/>
      </iq>

   S: <stream:error>
        <invalid-xml
            xmlns='urn:ietf:params:xml:ns:xmpp-streams'/>
      </stream:error>
      </stream:stream>
```

4.9.3.12. not-authorized

The entity has attempted to send XML stanzas or other outbound 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 data before sending the stream error.

(In the following example, the client attempts to send XML stanzas before authenticating with the server.)

```
   C: <?xml version='1.0'?>
      <stream:stream
          from='juliet@im.example.com'
          to='im.example.com'
          version='1.0'
          xmlns='jabber:client'
          xmlns:stream='http://etherx.jabber.org/streams'>

   S: <?xml version='1.0'?>
      <stream:stream
          from='im.example.com'
          id='++TR84Sm6A3hnt3Q065SnAbbk3Y='
          to='juliet@im.example.com'
          version='1.0'
          xml:lang='en'
          xmlns='jabber:client'
          xmlns:stream='http://etherx.jabber.org/streams'>

   C: <message to='romeo@example.net'>
        <body>Wherefore art thou?</body>
      </message>
   S: <stream:error>
        <not-authorized
            xmlns='urn:ietf:params:xml:ns:xmpp-streams'/>
      </stream:error>
      </stream:stream>
```

4.9.3.13. not-well-formed

The initiating entity has sent XML that violates the well-formedness rules of [XML] or [XML-NAMES].

(In the following example, the client sends an XMPP message that is not namespace-well-formed.)

```
   C: <message>
        <foo:body>What is this foo?</foo:body>
      </message>

   S: <stream:error>
        <not-well-formed
            xmlns='urn:ietf:params:xml:ns:xmpp-streams'/>
      </stream:error>
      </stream:stream>
```

Interoperability Note: In RFC 3920, the name of this error condition was "xml-not-well-formed" instead of "not-well-formed". The name was changed because the element name <xml-not-well-formed/> violates the constraint from Section 3 of [XML] that "names beginning with a match to (('X'|'x')('M'|'m')('L'|'l')) are reserved for standardization in this or future versions of this specification".

4.9.3.14. policy-violation

The entity has violated some local service policy (e.g., a stanza exceeds a configured size limit); the server MAY choose to specify the policy in the <text/> element or in an application-specific condition element.

(In the following example, the client sends an XMPP message that is too large according to the server's local service policy.)

```
   C: <message to='juliet@im.example.com' id='foo'>
        <body>[ ... the-emacs-manual ... ]</body>
      </message>
   S: <stream:error>
        <policy-violation
            xmlns='urn:ietf:params:xml:ns:xmpp-streams'/>
        <stanza-too-big xmlns='urn:xmpp:errors'/>
      </stream:error>

   S: </stream:stream>
```

4.9.3.15. remote-connection-failed

The server is unable to properly connect to a remote entity that is needed for authentication or authorization (e.g., in certain scenarios related to Server Dialback [XEP-0220]); this condition is not to be used when the cause of the error is within the administrative domain of the XMPP service provider, in which case the <internal-server-error/> condition is more appropriate.

```
   C: <?xml version='1.0'?>
      <stream:stream
          from='juliet@im.example.com'
          to='im.example.com'
          version='1.0'
          xmlns='jabber:client'
          xmlns:stream='http://etherx.jabber.org/streams'>

   S: <?xml version='1.0'?>
      <stream:stream
          from='im.example.com'
          id='++TR84Sm6A3hnt3Q065SnAbbk3Y='
          to='juliet@im.example.com'
          version='1.0'
          xml:lang='en'
          xmlns='jabber:client'
          xmlns:stream='http://etherx.jabber.org/streams'>
      <stream:error>
        <remote-connection-failed
            xmlns='urn:ietf:params:xml:ns:xmpp-streams'/>
      </stream:error>
      </stream:stream>
```

4.9.3.16. reset

The server is closing the stream because it has new (typically security-critical) features to offer, because the keys or certificates used to establish a secure context for the stream have expired or have been revoked during the life of the stream (Section 13.7.2.3), because the TLS sequence number has wrapped (Section 5.3.5), etc. The reset applies to the stream and to any security context established for that stream (e.g., via TLS and SASL), which means that encryption and authentication need to be negotiated again for the new stream (e.g., TLS session resumption cannot be used).

```
   S: <stream:error>
        <reset
            xmlns='urn:ietf:params:xml:ns:xmpp-streams'/>
      </stream:error>
      </stream:stream>
```

4.9.3.17. resource-constraint

The server lacks the system resources necessary to service the stream.

```
   C: <?xml version='1.0'?>
      <stream:stream
          from='juliet@im.example.com'
          to='im.example.com'
          version='1.0'
          xmlns='jabber:client'
          xmlns:stream='http://etherx.jabber.org/streams'>

   S: <?xml version='1.0'?>
      <stream:stream
          from='im.example.com'
          id='++TR84Sm6A3hnt3Q065SnAbbk3Y='
          to='juliet@im.example.com'
          version='1.0'
          xml:lang='en'
          xmlns='jabber:client'
          xmlns:stream='http://etherx.jabber.org/streams'>
      <stream:error>
        <resource-constraint
            xmlns='urn:ietf:params:xml:ns:xmpp-streams'/>
      </stream:error>
      </stream:stream>
```

4.9.3.18. restricted-xml

The entity has attempted to send restricted XML features such as a comment, processing instruction, DTD subset, or XML entity reference (see Section 11.1).

(In the following example, the client sends an XMPP message containing an XML comment.)

```
   C: <message to='juliet@im.example.com'>
        <!--<subject/>-->
        <body>This message has no subject.</body>
      </message>

   S: <stream:error>
        <restricted-xml
            xmlns='urn:ietf:params:xml:ns:xmpp-streams'/>
      </stream:error>
      </stream:stream>
```

4.9.3.19. see-other-host

The server will not provide service to the initiating entity but is redirecting traffic to another host under the administrative control of the same service provider. The XML character data of the <see-other-host/> element returned by the server MUST specify the alternate FQDN or IP address at which to connect, which MUST be a valid domainpart or a domainpart plus port number (separated by the ':' character in the form "domainpart:port"). If the domainpart is the same as the source domain, derived domain, or resolved IPv4 or IPv6 address to which the initiating entity originally connected (differing only by the port number), then the initiating entity SHOULD simply attempt to reconnect at that address. (The format of an IPv6 address MUST follow [IPv6-ADDR], which includes the enclosing the IPv6 address in square brackets '[' and ']' as originally defined by [URI].) Otherwise, the initiating entity MUST resolve the FQDN specified in the <see-other-host/> element as described under Section 3.2.

```
   C: <?xml version='1.0'?>
      <stream:stream
          from='juliet@im.example.com'
          to='im.example.com'
          version='1.0'
          xmlns='jabber:client'
          xmlns:stream='http://etherx.jabber.org/streams'>
   S: <?xml version='1.0'?>
      <stream:stream
          from='im.example.com'
          id='++TR84Sm6A3hnt3Q065SnAbbk3Y='
          to='juliet@im.example.com'
          version='1.0'
          xml:lang='en'
          xmlns='jabber:client'
          xmlns:stream='http://etherx.jabber.org/streams'>
      <stream:error>
        <see-other-host
           xmlns='urn:ietf:params:xml:ns:xmpp-streams'>
          [2001:41D0:1:A49b::1]:9222
        </see-other-host>
      </stream:error>
      </stream:stream>
```

When negotiating a stream with the host to which it has been redirected, the initiating entity MUST apply the same policies it would have applied to the original connection attempt (e.g., a policy requiring TLS), MUST specify the same 'to' address on the initial stream header, and MUST verify the identity of the new host using the same reference identifier(s) it would have used for the original connection attempt (in accordance with [TLS-CERTS]). Even if the receiving entity returns a <see-other-host/> error before the confidentiality and integrity of the stream have been established (thus introducing the possibility of a denial-of-service attack), the fact that the initiating entity needs to verify the identity of the XMPP service based on the same reference identifiers implies that the initiating entity will not connect to a malicious entity. To reduce the possibility of a denial-of-service attack, (a) the receiving entity SHOULD NOT close the stream with a <see-other-host/> stream error until after the confidentiality and integrity of the stream have been protected via TLS or an equivalent security layer (such as the SASL GSSAPI mechanism), and (b) the receiving entity MAY have a policy of following redirects only if it has authenticated the receiving entity. In addition, the initiating entity SHOULD abort the connection attempt after a certain number of successive redirects (e.g., at least 2 but no more than 5).

4.9.3.20. system-shutdown

The server is being shut down and all active streams are being closed.

```
   S: <stream:error>
        <system-shutdown
            xmlns='urn:ietf:params:xml:ns:xmpp-streams'/>
      </stream:error>
      </stream:stream>
```

4.9.3.21. undefined-condition

The error condition is not one of those defined by the other conditions in this list; this error condition SHOULD NOT be used except in conjunction with an application-specific condition.

```
   S: <stream:error>
        <undefined-condition
            xmlns='urn:ietf:params:xml:ns:xmpp-streams'/>
        <app-error xmlns='http://example.org/ns'/>
      </stream:error>
      </stream:stream>
```

4.9.3.22. unsupported-encoding

The initiating entity has encoded the stream in an encoding that is not supported by the server (see Section 11.6) or has otherwise improperly encoded the stream (e.g., by violating the rules of the [UTF-8] encoding).

(In the following example, the client attempts to encode data using UTF-16 instead of UTF-8.)

```
   C: <?xml version='1.0' encoding='UTF-16'?>
      <stream:stream
          from='juliet@im.example.com'
          to='im.example.com'
          version='1.0'
          xmlns='jabber:client'
          xmlns:stream='http://etherx.jabber.org/streams'>
   S: <?xml version='1.0'?>
      <stream:stream
          from='im.example.com'
          id='++TR84Sm6A3hnt3Q065SnAbbk3Y='
          to='juliet@im.example.com'
          version='1.0'
          xml:lang='en'
          xmlns='jabber:client'
          xmlns:stream='http://etherx.jabber.org/streams'>
      <stream:error>
        <unsupported-encoding
            xmlns='urn:ietf:params:xml:ns:xmpp-streams'/>
      </stream:error>
      </stream:stream>
```

4.9.3.23. unsupported-feature

The receiving entity has advertised a mandatory-to-negotiate stream feature that the initiating entity does not support, and has offered no other mandatory-to-negotiate feature alongside the unsupported feature.

(In the following example, the receiving entity requires negotiation of an example feature, but the initiating entity does not support the feature.)

```
   R: <stream:features>
        <example xmlns='urn:xmpp:example'>
          <required/>
        </example>
      </stream:features>

   I: <stream:error>
        <unsupported-feature
            xmlns='urn:ietf:params:xml:ns:xmpp-streams'/>
      </stream:error>
      </stream:stream>
```

4.9.3.24. unsupported-stanza-type

The initiating entity has sent a first-level child of the stream that is not supported by the server, either because the receiving entity does not understand the namespace or because the receiving entity does not understand the element name for the applicable namespace (which might be the content namespace declared as the default namespace).

(In the following example, the client attempts to send a first-level child element of <pubsub/> qualified by the 'jabber:client' namespace, but the schema for that namespace defines no such element.)

```
   C: <pubsub xmlns='jabber:client'>
        <publish node='princely_musings'>
          <item id='ae890ac52d0df67ed7cfdf51b644e901'>
            <entry xmlns='http://www.w3.org/2005/Atom'>
              <title>Soliloquy</title>
              <summary>
   To be, or not to be: that is the question:
   Whether 'tis nobler in the mind to suffer
   The slings and arrows of outrageous fortune,
   Or to take arms against a sea of troubles,
   And by opposing end them?
              </summary>
              <link rel='alternate' type='text/html'
                    href='http://denmark.example/2003/12/13/atom03'/>
              <id>tag:denmark.example,2003:entry-32397</id>
              <published>2003-12-13T18:30:02Z</published>
              <updated>2003-12-13T18:30:02Z</updated>
            </entry>
          </item>
        </publish>
      </pubsub>

   S: <stream:error>
        <unsupported-stanza-type
            xmlns='urn:ietf:params:xml:ns:xmpp-streams'/>
      </stream:error>
      </stream:stream>
```

4.9.3.25. unsupported-version

The 'version' attribute provided by the initiating entity in the stream header specifies a version of XMPP that is not supported by the server.

```
   C: <?xml version='1.0'?>
      <stream:stream
          from='juliet@im.example.com'
          to='im.example.com'
          version='11.0'
          xmlns='jabber:client'
          xmlns:stream='http://etherx.jabber.org/streams'>
   S: <?xml version='1.0'?>
      <stream:stream
          from='im.example.com'
          id='++TR84Sm6A3hnt3Q065SnAbbk3Y='
          to='juliet@im.example.com'
          version='1.0'
          xml:lang='en'
          xmlns='jabber:client'
          xmlns:stream='http://etherx.jabber.org/streams'>
      <stream:error>
        <unsupported-version
            xmlns='urn:ietf:params:xml:ns:xmpp-streams'/>
      </stream:error>
      </stream:stream>
```

4.9.4. 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.

```
   C: <message>
        <body>
          My keyboard layout is:

          QWERTYUIOP{}|
          ASDFGHJKL:"
          ZXCVBNM<>?
        </body>
      </message>

   S: <stream:error>
        <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='http://example.org/ns'/>
      </stream:error>
      </stream:stream>
```

4.10. Simplified Stream Examples

This section contains two highly simplified examples of a stream-based connection between a client and a server; these examples are included for the purpose of illustrating the concepts introduced thus far, but the reader needs to be aware that these examples elide many details (see Section 9 for more complete examples).

A basic connection:

```
   C: <?xml version='1.0'?>
      <stream:stream
          from='juliet@im.example.com'
          to='im.example.com'
          version='1.0'
          xml:lang='en'
          xmlns='jabber:client'
          xmlns:stream='http://etherx.jabber.org/streams'>

   S: <?xml version='1.0'?>
      <stream:stream
          from='im.example.com'
          id='++TR84Sm6A3hnt3Q065SnAbbk3Y='
          to='juliet@im.example.com'
          version='1.0'
          xml:lang='en'
          xmlns='jabber:client'
          xmlns:stream='http://etherx.jabber.org/streams'>

   [ ... stream negotiation ... ]

   C:   <message from='juliet@im.example.com/balcony'
                 to='romeo@example.net'
                 xml:lang='en'>
          <body>Art thou not Romeo, and a Montague?</body>
        </message>

   S:   <message from='romeo@example.net/orchard'
                 to='juliet@im.example.com/balcony'
                 xml:lang='en'>
          <body>Neither, fair saint, if either thee dislike.</body>
        </message>

   C: </stream:stream>

   S: </stream:stream>
```

A connection gone bad:

```
   C: <?xml version='1.0'?>
      <stream:stream
          from='juliet@im.example.com'
          to='im.example.com'
          version='1.0'
          xml:lang='en'
          xmlns='jabber:client'
          xmlns:stream='http://etherx.jabber.org/streams'>

   S: <?xml version='1.0'?>
      <stream:stream
          from='im.example.com'
          id='++TR84Sm6A3hnt3Q065SnAbbk3Y='
          to='juliet@im.example.com'
          version='1.0'
          xml:lang='en'
          xmlns='jabber:client'
          xmlns:stream='http://etherx.jabber.org/streams'>

   [ ... stream negotiation ... ]

   C:   <message from='juliet@im.example.com/balcony'
                 to='romeo@example.net'
                 xml:lang='en'>
          <body>No closing tag!
        </message>

   S: <stream:error>
       <not-well-formed
           xmlns='urn:ietf:params:xml:ns:xmpp-streams'/>
      </stream:error>
      </stream:stream>
```

More detailed examples are provided under Section 9.

5. STARTTLS Negotiation

5.1. Fundamentals

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, specifically a "STARTTLS" extension that is modeled after similar extensions for the [IMAP],

[POP3], and [ACAP] protocols as described in [USINGTLS]. The XML namespace name for the STARTTLS extension is 'urn:ietf:params:xml:ns:xmpp-tls'.

5.2. Support

Support for STARTTLS is REQUIRED in XMPP client and server implementations. An administrator of a given deployment MAY specify that TLS is mandatory-to-negotiate for client-to-server communication, server-to-server communication, or both. An initiating entity SHOULD use TLS to secure its stream with the receiving entity before proceeding with SASL authentication.

5.3. Stream Negotiation Rules

5.3.1. Mandatory-to-Negotiate

If the receiving entity advertises only the STARTTLS feature or if the receiving entity includes the <required/> child element as explained under Section 5.4.1, the parties MUST consider TLS as mandatory-to-negotiate. If TLS is mandatory-to-negotiate, the receiving entity SHOULD NOT advertise support for any stream feature except STARTTLS during the initial stage of the stream negotiation process, because further stream features might depend on prior negotiation of TLS given the order of layers in XMPP (e.g., the particular SASL mechanisms offered by the receiving entity will likely depend on whether TLS has been negotiated).

5.3.2. Restart

After TLS negotiation, the parties MUST restart the stream.

5.3.3. Data Formatting

During STARTTLS negotiation, the entities MUST NOT send any whitespace as separators between XML elements (i.e., from the last character of the first-level <starttls/> element qualified by the 'urn:ietf:params:xml:ns:xmpp-tls' namespace as sent by the initiating entity, until the last character of the first-level <proceed/> element qualified by the 'urn:ietf:params:xml:ns:xmpp-tls' namespace as sent by the receiving entity). This prohibition helps to ensure proper security layer byte precision. Any such whitespace shown in the STARTTLS examples provided in this document is included only for the sake of readability.

5.3.4. Order of TLS and SASL Negotiations

If the initiating entity chooses to use TLS, STARTTLS negotiation MUST be completed before proceeding to SASL negotiation (Section 6); this order of negotiation is necessary 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 (or other credentials) provided during prior TLS negotiation.

5.3.5. TLS Renegotiation

The TLS protocol allows either party in a TLS-protected channel to initiate a new handshake that establishes new cryptographic parameters (see [TLS-NEG]). The cases most commonly mentioned are:

1. Refreshing encryption keys.

2. Wrapping the TLS sequence number as explained in Section 6.1 of [TLS].

3. Protecting client credentials by completing server authentication first and then completing client authentication over the protected channel.

Because it is relatively inexpensive to establish streams in XMPP, for the first two cases it is preferable to use an XMPP stream reset (as described under Section 4.9.3.16) instead of performing TLS renegotiation.

The third case has improved security characteristics when the TLS client (which might be an XMPP server) presents credentials to the TLS server. If communicating such credentials to an unauthenticated TLS server might leak private information, it can be appropriate to complete TLS negotiation for the purpose of authenticating the TLS server to the TLS client and then attempt TLS renegotiation for the purpose of authenticating the TLS client to the TLS server. However, this case is extremely rare because the credentials presented by an XMPP server or XMPP client acting as a TLS client are almost always public (i.e., a PKIX certificate), and therefore providing those credentials before authenticating the XMPP server acting as a TLS server would not in general leak private information.

As a result, implementers are encouraged to carefully weigh the costs and benefits of TLS renegotiation before supporting it in their software, and XMPP entities that act as TLS clients are discouraged from attempting TLS renegotiation unless the certificate (or other credential information) sent during TLS negotiation is known to be private.

Support for TLS renegotiation is strictly OPTIONAL. However, implementations that support TLS renegotiation MUST implement and use the TLS Renegotiation Extension [TLS-NEG].

If an entity that does not support TLS renegotiation detects a renegotiation attempt, then it MUST immediately close the underlying TCP connection without returning a stream error (since the violation has occurred at the TLS layer, not the XMPP layer, as described under Section 13.3).

If an entity that supports TLS renegotiation detects a TLS renegotiation attempt that does not use the TLS Renegotiation Extension [TLS-NEG], then it MUST immediately close the underlying TCP connection without returning a stream error (since the violation has occurred at the TLS layer, not the XMPP layer as described under Section 13.3).

5.3.6. TLS Extensions

Either party to a stream MAY include any TLS extension during the TLS negotiation itself. This is a matter for the TLS layer, not the XMPP layer.

5.4. Process

5.4.1. Exchange of Stream Headers and Stream Features

The initiating entity resolves the FQDN of the receiving entity as specified under Section 3, opens a TCP connection to the advertised port at the resolved IP address, and sends an initial stream header to the receiving entity.

```
I: <stream:stream
     from='juliet@im.example.com'
     to='im.example.com'
     version='1.0'
     xml:lang='en'
     xmlns='jabber:client'
     xmlns:stream='http://etherx.jabber.org/streams'>
```

The receiving entity MUST send a response stream header to the initiating entity over the TCP connection opened by the initiating entity.

```
   R: <stream:stream
        from='im.example.com'
        id='t7AMCin9zjMNwQKDnplntZPIDEI='
        to='juliet@im.example.com'
        version='1.0'
        xml:lang='en'
        xmlns='jabber:client'
        xmlns:stream='http://etherx.jabber.org/streams'>
```

The receiving entity then MUST send stream features to the initiating entity. If the receiving entity supports TLS, the stream features MUST include an advertisement for support of STARTTLS negotiation, i.e., a <starttls/> element qualified by the 'urn:ietf:params:xml:ns:xmpp-tls' namespace.

If the receiving entity considers STARTTLS negotiation to be mandatory-to-negotiate, the <starttls/> element MUST contain an empty <required/> child element.

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

5.4.2. Initiation of STARTTLS Negotiation

5.4.2.1. STARTTLS Command

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

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

The receiving entity MUST reply with either a <proceed/> element (proceed case) or a <failure/> element (failure case) qualified by the 'urn:ietf:params:xml:ns:xmpp-tls' namespace.

5.4.2.2. Failure Case

If the failure case occurs, the receiving entity MUST return a <failure/> element qualified by the 'urn:ietf:params:xml:ns:xmpp-tls' namespace, close the XML stream, and terminate the underlying TCP connection.

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

R: </stream:stream>
```

Causes for the failure case include but are not limited to:

1. The initiating entity has sent a malformed STARTTLS command.
2. The receiving entity did not offer the STARTTLS feature in its stream features.
3. The receiving entity cannot complete STARTTLS negotiation because of an internal error.

> Informational Note: STARTTLS failure is not triggered by TLS errors such as bad_certificate or handshake_failure, which are generated and handled during the TLS negotiation itself as described in [TLS].

If the failure case occurs, the initiating entity MAY attempt to reconnect as explained under Section 3.3.

5.4.2.3. Proceed Case

If the proceed case occurs, the receiving entity MUST return a <proceed/> element qualified by the 'urn:ietf:params:xml:ns:xmpp-tls' namespace.

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

The receiving entity MUST consider the TLS negotiation to have begun immediately after sending the closing '>' character of the <proceed/> element to the initiating entity. The initiating entity MUST consider the TLS negotiation to have begun immediately after receiving the closing '>' character of the <proceed/> element from the receiving entity.

The entities now proceed to TLS negotiation as explained in the next section.

5.4.3. TLS Negotiation

5.4.3.1. Rules

In order to complete TLS negotiation over the TCP connection, the entities MUST follow the process defined in [TLS].

The following rules apply:

1. The entities MUST NOT send any further XML data until the TLS negotiation is complete.

2. When using any of the mandatory-to-implement (MTI) ciphersuites specified under Section 13.8, the receiving entity MUST present a certificate.

3. So that mutual certificate authentication will be possible, the receiving entity SHOULD send a certificate request to the initiating entity, and the initiating entity SHOULD send a certificate to the receiving entity (but for privacy reasons might opt not to send a certificate until after the receiving entity has authenticated to the initiating entity).

4. The receiving entity SHOULD choose which certificate to present based on the domainpart contained in the 'to' attribute of the initial stream header (in essence, this domainpart is functionally equivalent to the Server Name Indication defined for TLS in [TLS-EXT]).

5. To determine if the TLS negotiation will succeed, the initiating entity MUST attempt to validate the receiving entity's certificate in accordance with the certificate validation procedures specified under Section 13.7.2.

6. If the initiating entity presents a certificate, the receiving entity too MUST attempt to validate the initiating entity's certificate in accordance with the certificate validation procedures specified under Section 13.7.2.

7. Following successful TLS negotiation, all further data transmitted by either party MUST be protected with the negotiated algorithms, keys, and secrets (i.e., encrypted, integrity-protected, or both depending on the ciphersuite used).

   Security Warning: See Section 13.8 regarding ciphersuites that MUST be supported for TLS; naturally, other ciphersuites MAY be supported as well.

5.4.3.2. TLS Failure

If the TLS negotiation results in failure, the receiving entity MUST terminate the TCP connection.

The receiving entity MUST NOT send a closing </stream> tag before terminating the TCP connection (since the failure has occurred at the TLS layer, not the XMPP layer as described under Section 13.3).

The initiating entity MAY attempt to reconnect as explained under Section 3.3, with or without attempting TLS negotiation (in accordance with local service policy, user-configured preferences, etc.).

5.4.3.3. TLS Success

If the TLS negotiation is successful, then the entities MUST proceed as follows.

1. The initiating entity MUST discard any information transmitted in layers above TCP that it obtained from the receiving entity in an insecure manner before TLS took effect (e.g., the receiving entity's 'from' address or the stream ID and stream features received from the receiving entity).

2. The receiving entity MUST discard any information transmitted in layers above TCP that it obtained from the initiating entity in an insecure manner before TLS took effect (e.g., the initiating entity's 'from' address).

3. The initiating entity MUST send a new initial stream header to the receiving entity over the encrypted connection (as specified under Section 4.3.3, the initiating entity MUST NOT send a closing </stream> tag before sending the new initial stream header, since the receiving entity and initiating entity MUST consider the original stream to be replaced upon success of the TLS negotiation).

```
I: <stream:stream
     from='juliet@im.example.com'
     to='im.example.com'
     version='1.0'
     xml:lang='en'
     xmlns='jabber:client'
     xmlns:stream='http://etherx.jabber.org/streams'>
```

4. The receiving entity MUST respond with a new response stream header over the encrypted connection (for which it MUST generate a new stream ID instead of reusing the old stream ID).

```
   R: <stream:stream
        from='im.example.com'
        id='vgKi/bkYME8OAj4rlXMkpucAqe4='
        to='juliet@im.example.com'
        version='1.0'
        xml:lang='en'
        xmlns='jabber:client'
        xmlns:stream='http://etherx.jabber.org/streams'>
```

5. The receiving entity also MUST send stream features to the initiating entity, which MUST NOT include the STARTTLS feature but which SHOULD include the SASL stream feature as described under Section 6 (see especially Section 6.4.1 regarding the few reasons why the SASL stream feature would not be offered here).

```
   R: <stream:features>
        <mechanisms xmlns='urn:ietf:params:xml:ns:xmpp-sasl'>
          <mechanism>EXTERNAL</mechanism>
          <mechanism>SCRAM-SHA-1-PLUS</mechanism>
          <mechanism>SCRAM-SHA-1</mechanism>
          <mechanism>PLAIN</mechanism>
        </mechanisms>
      </stream:features>
```

6. SASL Negotiation

6.1. Fundamentals

XMPP includes a method for authenticating a stream by means of an XMPP-specific profile of the Simple Authentication and Security Layer protocol (see [SASL]). SASL provides a generalized method for adding authentication support to connection-based protocols, and XMPP uses an XML namespace profile of SASL that conforms to the profiling requirements of [SASL]. The XML namespace name for the SASL extension is 'urn:ietf:params:xml:ns:xmpp-sasl'.

6.2. Support

Support for SASL negotiation is REQUIRED in XMPP client and server implementations.

6.3. Stream Negotiation Rules

6.3.1. Mandatory-to-Negotiate

The parties to a stream MUST consider SASL as mandatory-to-negotiate.

6.3.2. Restart

After SASL negotiation, the parties MUST restart the stream.

6.3.3. Mechanism Preferences

Any entity that will act as a SASL client or a SASL server MUST maintain an ordered list of its preferred SASL mechanisms according to the client or server, where the list is ordered according to local policy or user configuration (which SHOULD be in order of perceived strength to enable the strongest authentication possible). The initiating entity MUST maintain its own preference order independent of the preference order of the receiving entity. A client MUST try SASL mechanisms in its preference order. For example, if the server offers the ordered list "PLAIN SCRAM-SHA-1 GSSAPI" or "SCRAM-SHA-1 GSSAPI PLAIN" but the client's ordered list is "GSSAPI SCRAM-SHA-1", the client MUST try GSSAPI first and then SCRAM-SHA-1 but MUST NOT try PLAIN (since PLAIN is not on its list).

6.3.4. Mechanism Offers

If the receiving entity considers TLS negotiation (Section 5) to be mandatory-to-negotiate before it will accept authentication with a particular SASL mechanism, it MUST NOT advertise that mechanism in its list of available SASL mechanisms before TLS negotiation has been completed.

The receiving entity SHOULD offer the SASL EXTERNAL mechanism if both of the following conditions hold:

1. During TLS negotiation the initiating entity presented a certificate that is acceptable to the receiving entity for purposes of strong identity verification in accordance with local service policies (e.g., because said certificate is unexpired, is unrevoked, and is anchored to a root trusted by the receiving entity).

2. The receiving entity expects that the initiating entity will be able to authenticate and authorize as the identity provided in the certificate; in the case of a server-to-server stream, the receiving entity might have such an expectation because a DNS domain name presented in the initiating entity's certificate matches the domain referenced in the 'from' attribute of the initial stream header, where the matching rules of [TLS-CERTS] apply; in the case of a client-to-server stream, the receiving entity might have such an expectation because the bare JID presented in the initiating entity's certificate matches a user account that is registered with the server or because other

information contained in the initiating entity's certificate matches that of an entity that has permission to use the server for access to an XMPP network.

However, the receiving entity MAY offer the SASL EXTERNAL mechanism under other circumstances, as well.

When the receiving entity offers the SASL EXTERNAL mechanism, the receiving entity SHOULD list the EXTERNAL mechanism first among its offered SASL mechanisms and the initiating entity SHOULD attempt SASL negotiation using the EXTERNAL mechanism first (this preference will tend to increase the likelihood that the parties can negotiate mutual certificate authentication).

Section 13.8 specifies SASL mechanisms that MUST be supported; naturally, other SASL mechanisms MAY be supported as well.

Informational Note: Best practices for the use of SASL in the context of XMPP are described in [XEP-0175] for the ANONYMOUS mechanism and in [XEP-0178] for the EXTERNAL mechanism.

6.3.5. Data Formatting

The following data formatting rules apply to the SASL negotiation:

1. During SASL negotiation, the entities MUST NOT send any whitespace as separators between XML elements (i.e., from the last character of the first-level <auth/> element qualified by the 'urn:ietf:params:xml:ns:xmpp-sasl' namespace as sent by the initiating entity, until the last character of the first-level <success/> element qualified by the 'urn:ietf:params:xml:ns:xmpp-sasl' namespace as sent by the receiving entity). This prohibition helps to ensure proper security layer byte precision. Any such whitespace shown in the SASL examples provided in this document is included only for the sake of readability.

2. Any XML character data contained within the XML elements MUST be encoded using base 64, where the encoding adheres to the definition in Section 4 of [BASE64] and where the padding bits are set to zero.

3. As formally specified in the XML schema for the 'urn:ietf:params:xml:ns:xmpp-sasl' namespace under Appendix A.4, the receiving entity MAY include one or more application-specific child elements inside the <mechanisms/> element to provide information that might be needed by the initiating entity in order to complete successful SASL negotiation using one or more

of the offered mechanisms; however, the syntax and semantics of all such elements are out of scope for this specification (see [XEP-0233] for one example).

6.3.6. Security Layers

Upon successful SASL negotiation that involves negotiation of a security layer, both the initiating entity and the receiving entity MUST discard any application-layer state (i.e, state from the XMPP layer, excluding state from the TLS negotiation or SASL negotiation).

6.3.7. Simple User Name

Some SASL mechanisms (e.g., CRAM-MD5, DIGEST-MD5, and SCRAM) specify that the authentication identity used in the context of such mechanisms is a "simple user name" (see Section 2 of [SASL] as well as [SASLPREP]). The exact form of the simple user name in any particular mechanism or deployment thereof is a local matter, and a simple user name does not necessarily map to an application identifier such as a JID or JID component (e.g., a localpart). However, in the absence of local information provided by the server, an XMPP client SHOULD assume that the authentication identity for such a SASL mechanism is a simple user name equal to the localpart of the user's JID.

6.3.8. Authorization Identity

An authorization identity is an OPTIONAL identity included by the initiating entity to specify an identity to act as (see Section 2 of [SASL]). In client-to-server streams, it would most likely be used by an administrator to perform some management task on behalf of another user, whereas in server-to-server streams it would most likely be used to specify a particular add-on service at an XMPP service (e.g., a multi-user chat server at conference.example.com that is hosted by the example.com XMPP service). 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.

In the case of client-to-server communication, the value of an authorization identity MUST be a bare JID (<localpart@domainpart>) rather than a full JID (<localpart@domainpart/resourcepart>).

In the case of server-to-server communication, the value of an authorization identity MUST be a domainpart only (<domainpart>).

If the initiating entity provides an authorization identity during SASL negotiation, the receiving entity is responsible for verifying that the initiating entity is in fact allowed to assume the specified authorization identity; if not, the receiving entity MUST return an <invalid-authzid/> SASL error as described under Section 6.5.6.

6.3.9. Realms

The receiving entity MAY include a realm when negotiating certain SASL mechanisms (e.g., both the GSSAPI and DIGEST-MD5 mechanisms allow the authentication exchange to include a realm, though in different ways, whereas the EXTERNAL, SCRAM, and PLAIN mechanisms do not). If the receiving entity does not communicate a realm, the initiating entity MUST NOT assume that any realm exists. The realm MUST be used only for the purpose of authentication; in particular, an initiating entity MUST NOT attempt to derive an XMPP domainpart from the realm information provided by the receiving entity.

6.3.10. Round Trips

[SASL] specifies that a using protocol such as XMPP can define two methods by which the protocol can save round trips where allowed for the SASL mechanism:

1. When the SASL client (the XMPP "initiating entity") requests an authentication exchange, it can include "initial response" data with its request if appropriate for the SASL mechanism in use. In XMPP, this is done by including the initial response as the XML character data of the <auth/> element.

2. At the end of the authentication exchange, the SASL server (the XMPP "receiving entity") can include "additional data with success" if appropriate for the SASL mechanism in use. In XMPP, this is done by including the additional data as the XML character data of the <success/> element.

For the sake of protocol efficiency, it is REQUIRED for clients and servers to support these methods and RECOMMENDED to use them; however, clients and servers MUST support the less efficient modes as well.

6.4. Process

The process for SASL negotiation is as follows.

6.4.1. Exchange of Stream Headers and Stream Features

If SASL negotiation follows successful STARTTLS negotiation (Section 5), then the SASL negotiation occurs over the protected stream that has already been negotiated. If not, the initiating entity resolves the FQDN of the receiving entity as specified under Section 3, opens a TCP connection to the advertised port at the resolved IP address, and sends an initial stream header to the receiving entity. In either case, the receiving entity will receive an initial stream from the initiating entity.

```
I: <stream:stream
     from='juliet@im.example.com'
     to='im.example.com'
     version='1.0'
     xml:lang='en'
     xmlns='jabber:client'
     xmlns:stream='http://etherx.jabber.org/streams'>
```

When the receiving entity processes an initial stream header from the initiating entity, it MUST send a response stream header to the initiating entity (for which it MUST generate a unique stream ID. If TLS negotiation has already succeeded, then this stream ID MUST be different from the stream ID sent before TLS negotiation succeeded).

```
R: <stream:stream
     from='im.example.com'
     id='vgKi/bkYME8OAj4rlXMkpucAqe4='
     to='juliet@im.example.com'
     version='1.0'
     xml:lang='en'
     xmlns='jabber:client'
     xmlns:stream='http://etherx.jabber.org/streams'>
```

The receiving entity also MUST send stream features to the initiating entity. The stream features SHOULD include an advertisement for support of SASL negotiation, i.e., a <mechanisms/> element qualified by the 'urn:ietf:params:xml:ns:xmpp-sasl' namespace. Typically there are only three cases in which support for SASL negotiation would not be advertised here:

- TLS negotiation needs to happen before SASL can be offered (i.e., TLS is required and the receiving entity is responding to the very first initial stream header it has received for this connection attempt).

- SASL negotiation is impossible for a server-to-server connection (i.e., the initiating server has not provided a certificate that would enable strong authentication and therefore the receiving server is falling back to weak identity verification using the Server Dialback protocol [XEP-0220]).

- SASL has already been negotiated (i.e., the receiving entity is responding to an initial stream header sent as a stream restart after successful SASL negotiation).

The <mechanisms/> element MUST contain one <mechanism/> child element for each authentication mechanism the receiving entity offers to the initiating entity. As noted, the order of <mechanism/> elements in the XML indicates the preference order of the SASL mechanisms according to the receiving entity (which is not necessarily the preference order according to the initiating entity).

```
R: <stream:features>
     <mechanisms xmlns='urn:ietf:params:xml:ns:xmpp-sasl'>
       <mechanism>EXTERNAL</mechanism>
       <mechanism>SCRAM-SHA-1-PLUS</mechanism>
       <mechanism>SCRAM-SHA-1</mechanism>
       <mechanism>PLAIN</mechanism>
     </mechanisms>
   </stream:features>
```

6.4.2. Initiation

In order to begin the SASL negotiation, the initiating entity sends an <auth/> element qualified by the 'urn:ietf:params:xml:ns:xmpp-sasl' namespace and includes an appropriate value for the 'mechanism' attribute, thus starting the handshake for that particular authentication mechanism. 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 character ("="), which indicates that the response is present but contains no data.

```
I: <auth xmlns='urn:ietf:params:xml:ns:xmpp-sasl'
         mechanism='PLAIN'>AGp1bGlldAByMG0zMG15cjBtMzA=</auth>
```

If the initiating entity subsequently sends another <auth/> element and the ongoing authentication handshake has not yet completed, the receiving entity MUST discard the ongoing handshake and MUST process a new handshake for the subsequently requested SASL mechanism.

6.4.3. Challenge-Response Sequence

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; this element MAY contain XML character data (which MUST be generated in accordance with the definition of the SASL mechanism chosen by the initiating entity).

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

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:

- o The initiating entity aborts the handshake for this authentication mechanism.
- o The receiving entity reports failure of the handshake.
- o The receiving entity reports success of the handshake.

These scenarios are described in the following sections.

6.4.4. Abort

The initiating entity aborts the handshake for this authentication mechanism by sending an <abort/> element qualified by the 'urn:ietf:params:xml:ns:xmpp-sasl' namespace.

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

Upon receiving an <abort/> element, the receiving entity MUST return a <failure/> element qualified by the 'urn:ietf:params:xml:ns:xmpp-sasl' namespace and containing an <aborted/> child element.

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

6.4.5. SASL Failure

The receiving entity reports failure of the handshake for this authentication mechanism by sending a <failure/> element qualified by the 'urn:ietf:params:xml:ns:xmpp-sasl' namespace (the particular cause of failure MUST be communicated in an appropriate child element of the <failure/> element as defined under Section 6.5).

```
R: <failure xmlns='urn:ietf:params:xml:ns:xmpp-sasl'>
     <not-authorized/>
   </failure>
```

Where appropriate for the chosen SASL mechanism, the receiving entity SHOULD allow a configurable but reasonable number of retries (at least 2 and no more than 5); 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 (which it would if the receiving entity immediately returned a SASL failure and closed the stream).

If the initiating entity attempts a reasonable number of retries with the same SASL mechanism and all attempts fail, it MAY fall back to the next mechanism in its ordered list by sending a new <auth/> request to the receiving entity, thus starting a new handshake for that authentication mechanism. If all handshakes fail and there are no remaining mechanisms in the initiating entity's list of supported and acceptable mechanisms, the initiating entity SHOULD simply close the stream as described under Section 4.4 (instead of waiting for the stream to time out).

If the initiating entity exceeds the number of retries, the receiving entity MUST close the stream with a stream error, which SHOULD be <policy-violation/> (Section 4.9.3.14), although some existing implementations send <not-authorized/> (Section 4.9.3.12) instead.

Implementation Note: For server-to-server streams, if the receiving entity cannot offer the SASL EXTERNAL mechanism or any other SASL mechanism based on the security context established during TLS negotiation, the receiving entity MAY attempt to complete weak identity verification using the Server Dialback protocol [XEP-0220]; however, if according to local service policies weak identity verification is insufficient then the receiving entity SHOULD instead close the stream with a <policy-violation/> stream error (Section 4.9.3.14) instead of waiting for the stream to time out.

6.4.6. SASL Success

Before considering the SASL handshake to be a success, if the initiating entity provided a 'from' attribute on an initial stream header whose confidentiality and integrity were protected via TLS or an equivalent security layer (such as the SASL GSSAPI mechanism) then the receiving entity SHOULD correlate the authentication identity resulting from the SASL negotiation with that 'from' address; if the two identities do not match then the receiving entity SHOULD terminate the connection attempt (however, the receiving entity might have legitimate reasons not to terminate the connection attempt, for example, because it has overridden a connecting client's address to correct the JID format or assign a JID based on information presented in an end-user certificate).

The receiving entity reports success of the handshake by sending a <success/> element qualified by the 'urn:ietf:params:xml:ns:xmpp-sasl' namespace; this element MAY contain XML character data (in SASL terminology, "additional data with success") if the chosen SASL mechanism supports or requires it. If the receiving entity needs to send additional data of zero length, it MUST transmit the data as a single equals sign character ("=").

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

Informational Note: For client-to-server streams, the authorization identity communicated during SASL negotiation is used to determine the canonical address for the initiating client according to the receiving server, as described under Section 4.3.6.

Upon receiving the <success/> element, the initiating entity MUST initiate a new stream over the existing TCP connection by sending a new initial stream header to the receiving entity (as specified under Section 4.3.3, the initiating entity MUST NOT send a closing </stream> tag before sending the new initial stream header, since the receiving entity and initiating entity MUST consider the original stream to be replaced upon success of the SASL negotiation).

```
   I: <stream:stream
        from='juliet@im.example.com'
        to='im.example.com'
        version='1.0'
        xml:lang='en'
        xmlns='jabber:client'
        xmlns:stream='http://etherx.jabber.org/streams'>
```

Upon receiving the new initial stream header from the initiating entity, the receiving entity MUST respond by sending a new response stream header to the initiating entity (for which it MUST generate a new stream ID instead of reusing the old stream ID).

```
   R: <stream:stream
        from='im.example.com'
        id='gPybzaOzBmaADgxKXu9UClbprp0='
        to='juliet@im.example.com'
        version='1.0'
        xml:lang='en'
        xmlns='jabber:client'
        xmlns:stream='http://etherx.jabber.org/streams'>
```

The receiving entity MUST also send stream features, containing any further available features or containing no features (via an empty <features/> element).

```
   R: <stream:features>
        <bind xmlns='urn:ietf:params:xml:ns:xmpp-bind'/>
      </stream:features>
```

6.5. SASL Errors

The syntax of SASL errors is as follows, where the XML data shown within the square brackets '[' and ']' is OPTIONAL.

```
   <failure xmlns='urn:ietf:params:xml:ns:xmpp-sasl'>
     <defined-condition/>
     [<text xml:lang='langcode'>
         OPTIONAL descriptive text
     </text>]
   </failure>
```

The "defined-condition" MUST be one of the SASL-related error conditions defined in the following sections. However, because additional error conditions might be defined in the future, if an entity receives a SASL error condition that it does not understand then it MUST treat the unknown condition as a generic authentication failure, i.e., as equivalent to <not-authorized/> (Section 6.5.10).

Inclusion of the <text/> element is OPTIONAL, and can be used to provide application-specific information about the error condition, which information MAY be displayed to a human but only as a supplement to the defined condition.

Because XMPP itself defines an application profile of SASL and there is no expectation that more specialized XMPP applications will be built on top of SASL, the SASL error format does not provide extensibility for application-specific error conditions as is done for XML streams (Section 4.9.4) and XML stanzas (Section 8.3.4).

6.5.1. aborted

The receiving entity acknowledges that the authentication handshake has been aborted by the initiating entity; sent in reply to the <abort/> element.

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

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

6.5.2. account-disabled

The account of the initiating entity has been temporarily disabled; sent in reply to an <auth/> element (with or without initial response data) or a <response/> element.

```
I: <auth xmlns='urn:ietf:params:xml:ns:xmpp-sasl'
         mechanism='PLAIN'>AGp1bGlldAByMG0zMG15cjBtMzA=</auth>

R: <failure xmlns='urn:ietf:params:xml:ns:xmpp-sasl'>
     <account-disabled/>
     <text xml:lang='en'>Call 212-555-1212 for assistance.</text>
   </failure>
```

6.5.3. credentials-expired

The authentication failed because the initiating entity provided credentials that have expired; sent in reply to a <response/> element or an <auth/> element with initial response data.

```
I: <response xmlns='urn:ietf:params:xml:ns:xmpp-sasl'>
     [ ... ]
   </response>
R: <failure xmlns='urn:ietf:params:xml:ns:xmpp-sasl'>
     <credentials-expired/>
   </failure>
```

6.5.4. encryption-required

The mechanism requested by the initiating entity cannot be used unless the confidentiality and integrity of the underlying stream are protected (typically via TLS); sent in reply to an <auth/> element (with or without initial response data).

```
   I: <auth xmlns='urn:ietf:params:xml:ns:xmpp-sasl'
            mechanism='PLAIN'>AGp1bGlldAByMG0zMG15cjBtMzA=</auth>

   R: <failure xmlns='urn:ietf:params:xml:ns:xmpp-sasl'>
        <encryption-required/>
      </failure>
```

6.5.5. incorrect-encoding

The data provided by the initiating entity could not be processed because the base 64 encoding is incorrect (e.g., because the encoding does not adhere to the definition in Section 4 of [BASE64]); sent in reply to a <response/> element or an <auth/> element with initial response data.

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

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

6.5.6. invalid-authzid

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 <response/> element or an <auth/> element with initial response data.

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

   R: <failure xmlns='urn:ietf:params:xml:ns:xmpp-sasl'>
        <invalid-authzid/>
      </failure>
```

6.5.7. invalid-mechanism

The initiating entity did not specify a mechanism, or requested a mechanism that is not supported by the receiving entity; sent in reply to an <auth/> element.

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

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

6.5.8. malformed-request

The request is malformed (e.g., the <auth/> element includes initial response data but the mechanism does not allow that, or the data sent violates the syntax for the specified SASL mechanism); sent in reply to an <abort/>, <auth/>, <challenge/>, or <response/> element.

(In the following example, the XML character data of the <auth/> element contains more than 255 UTF-8-encoded Unicode characters and therefore violates the "token" production for the SASL ANONYMOUS mechanism as specified in [ANONYMOUS].)

```
I: <auth xmlns='urn:ietf:params:xml:ns:xmpp-sasl'
         mechanism='ANONYMOUS'>[ ... some-long-token ... ]</auth>

R: <failure xmlns='urn:ietf:params:xml:ns:xmpp-sasl'>
     <malformed-request/>
   </failure>
```

6.5.9. mechanism-too-weak

The mechanism requested by the initiating entity is weaker than server policy permits for that initiating entity; sent in reply to an <auth/> element (with or without initial response data).

```
I: <auth xmlns='urn:ietf:params:xml:ns:xmpp-sasl'
         mechanism='PLAIN'>AGp1bGlldAByMG0zMG15cjBtMzA=</auth>

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

6.5.10. not-authorized

The authentication failed because the initiating entity did not provide proper credentials, or because some generic authentication failure has occurred but the receiving entity does not wish to disclose specific information about the cause of the failure; sent in reply to a <response/> element or an <auth/> element with initial response data.

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

R: <failure xmlns='urn:ietf:params:xml:ns:xmpp-sasl'>
     <not-authorized/>
   </failure>
```

Security Warning: This error condition includes but is not limited to the case of incorrect credentials or a nonexistent username. In order to discourage directory harvest attacks, no differentiation is made between incorrect credentials and a nonexistent username.

6.5.11. temporary-auth-failure

The authentication failed because of a temporary error condition within the receiving entity, and it is advisable for the initiating entity to try again later; sent in reply to an <auth/> element or a <response/> element.

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

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

6.6. SASL Definition

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

service name: "xmpp"

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

exchange sequence: Challenges and responses are carried through the exchange of <challenge/> elements from receiving entity to initiating entity and <response/> elements from initiating entity to receiving entity. The receiving entity reports failure by sending a <failure/> element and success by sending a <success/> element; the initiating entity aborts the exchange by sending an <abort/> element. 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 <success/> element for the receiving entity, and immediately after receiving the closing '>' character of the <success/> 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 can be used in XMPP to denote the non-default <localpart@domainpart> of a client; an empty string is equivalent to an absent authorization identity.

7. Resource Binding

7.1. Fundamentals

After a client authenticates with a server, it MUST bind a specific resource to the stream so that the server can properly address the client. That is, there MUST be an XMPP resource associated with the bare JID (<localpart@domainpart>) of the client, so that the address for use over that stream is a full JID of the form <localpart@domainpart/resource> (including the resourcepart). This ensures that the server can deliver XML stanzas to and receive XML stanzas from the client in relation to entities other than the server itself or the client's account, as explained under Section 10.

Informational Note: The client could exchange stanzas with the server itself or the client's account before binding a resource since the full JID is needed only for addressing outside the context of the stream negotiated between the client and the server, but this is not commonly done.

After a client has bound a resource to the stream, it is referred to as a "connected resource". A server SHOULD allow an entity to maintain multiple connected resources simultaneously, where each connected resource is associated with a distinct XML stream and is differentiated from the other connected resources by a distinct resourcepart.

> Security Warning: A server SHOULD enable the administrator of an XMPP service to limit the number of connected resources in order to prevent certain denial-of-service attacks as described under Section 13.12.

If, before completing the resource binding step, the client attempts to send an XML stanza to an entity other than the server itself or the client's account, the server MUST NOT process the stanza and MUST close the stream with a <not-authorized/> stream error (Section 4.9.3.12).

The XML namespace name for the resource binding extension is 'urn:ietf:params:xml:ns:xmpp-bind'.

7.2. Support

Support for resource binding is REQUIRED in XMPP client and server implementations.

7.3. Stream Negotiation Rules

7.3.1. Mandatory-to-Negotiate

The parties to a stream MUST consider resource binding as mandatory-to-negotiate.

7.3.2. Restart

After resource binding, the parties MUST NOT restart the stream.

7.4. Advertising Support

Upon sending a new response stream header to the client after successful SASL negotiation, the server MUST include a <bind/> element qualified by the 'urn:ietf:params:xml:ns:xmpp-bind' namespace in the stream features it presents to the client.

The server MUST NOT include the resource binding stream feature until after the client has authenticated, typically by means of successful SASL negotiation.

```
   S: <stream:stream
          from='im.example.com'
          id='gPybzaOzBmaADgxKXu9UClbprp0='
          to='juliet@im.example.com'
          version='1.0'
          xml:lang='en'
          xmlns='jabber:client'
          xmlns:stream='http://etherx.jabber.org/streams'>

   S: <stream:features>
        <bind xmlns='urn:ietf:params:xml:ns:xmpp-bind'/>
      </stream:features>
```

Upon being informed that resource binding is mandatory-to-negotiate, the client MUST bind a resource to the stream as described in the following sections.

7.5. Generation of Resource Identifiers

A resourcepart MUST at a minimum be unique among the connected resources for that <localpart@domainpart>. Enforcement of this policy is the responsibility of the server.

Security Warning: A resourcepart can be security-critical. For example, if a malicious entity can guess a client's resourcepart then it might be able to determine if the client (and therefore the controlling principal) is online or offline, thus resulting in a presence leak as described under Section 13.10.2. To prevent that possibility, a client can either (1) generate a random resourcepart on its own or (2) ask the server to generate a resourcepart on its behalf. One method for ensuring that the resourcepart is random is to generate a Universally Unique Identifier (UUID) as specified in [UUID].

7.6. Server-Generated Resource Identifier

A server MUST be able to generate an XMPP resourcepart on behalf of a client. The resourcepart generated by the server MUST be random (see [RANDOM]).

7.6.1. Success Case

A client requests a server-generated resourcepart by sending an IQ stanza of type "set" (see Section 8.2.3) containing an empty <bind/> element qualified by the 'urn:ietf:params:xml:ns:xmpp-bind' namespace.

```
   C: <iq id='tn281v37' type='set'>
       <bind xmlns='urn:ietf:params:xml:ns:xmpp-bind'/>
      </iq>
```

Once the server has generated an XMPP resourcepart for 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.

```
   S: <iq id='tn281v37' type='result'>
       <bind xmlns='urn:ietf:params:xml:ns:xmpp-bind'>
         <jid>
           juliet@im.example.com/4db06f06-1ea4-11dc-aca3-000bcd821bfb
         </jid>
       </bind>
      </iq>
```

7.6.2. Error Cases

When a client asks the server to generate a resourcepart during resource binding, the following stanza error conditions are defined:

- o The account has reached a limit on the number of simultaneous connected resources allowed.

- o The client is otherwise not allowed to bind a resource to the stream.

Naturally, it is possible that error conditions not specified here might occur, as described under Section 8.3.

7.6.2.1. Resource Constraint

If the account has reached a limit on the number of simultaneous connected resources allowed, the server MUST return a <resource-constraint/> stanza error (Section 8.3.3.18).

```
   S: <iq id='tn281v37' type='error'>
        <error type='wait'>
          <resource-constraint
              xmlns='urn:ietf:params:xml:ns:xmpp-stanzas'/>
        </error>
      </iq>
```

7.6.2.2. Not Allowed

If the client is otherwise not allowed to bind a resource to the stream, the server MUST return a <not-allowed/> stanza error (Section 8.3.3.10).

```
S: <iq id='tn281v37' type='error'>
     <error type='cancel'>
       <not-allowed
           xmlns='urn:ietf:params:xml:ns:xmpp-stanzas'/>
     </error>
   </iq>
```

7.7. Client-Submitted Resource Identifier

Instead of asking the server to generate a resourcepart on its behalf, a client MAY attempt to submit a resourcepart that it has generated or that the controlling user has provided.

7.7.1. Success Case

A client asks its server to accept a client-submitted resourcepart by sending an IQ stanza of type "set" containing a <bind/> element with a child <resource/> element containing non-zero-length XML character data.

```
C: <iq id='wy2xa82b4' type='set'>
     <bind xmlns='urn:ietf:params:xml:ns:xmpp-bind'>
       <resource>balcony</resource>
     </bind>
   </iq>
```

The server SHOULD accept the client-submitted resourcepart. It does so by returning an IQ stanza of type "result" to the client, including a <jid/> child element that specifies the full JID for the connected resource and contains without modification the client-submitted text.

```
S: <iq id='wy2xa82b4' type='result'>
    <bind xmlns='urn:ietf:params:xml:ns:xmpp-bind'>
      <jid>juliet@im.example.com/balcony</jid>
    </bind>
   </iq>
```

Alternatively, in accordance with local service policies the server MAY refuse the client-submitted resourcepart and override it with a resourcepart that the server generates.

```
   S: <iq id='wy2xa82b4' type='result'>
       <bind xmlns='urn:ietf:params:xml:ns:xmpp-bind'>
         <jid>
      juliet@im.example.com/balcony 4db06f06-1ea4-11dc-aca3-000bcd821bfb
         </jid>
       </bind>
      </iq>
```

7.7.2. Error Cases

When a client attempts to submit its own XMPP resourcepart during resource binding, the following stanza error conditions are defined in addition to those described under Section 7.6.2:

o The provided resourcepart cannot be processed by the server.

o The provided resourcepart is already in use.

Naturally, it is possible that error conditions not specified here might occur, as described under Section 8.3.

7.7.2.1. Bad Request

If the provided resourcepart cannot be processed by the server (e.g., because it is of zero length or because it otherwise violates the rules for resourceparts specified in [XMPP-ADDR]), the server can return a <bad-request/> stanza error (Section 8.3.3.1) but SHOULD instead process the resourcepart so that it is in conformance.

```
   S: <iq id='wy2xa82b4' type='error'>
        <error type='modify'>
          <bad-request xmlns='urn:ietf:params:xml:ns:xmpp-stanzas'/>
        </error>
      </iq>
```

7.7.2.2. Conflict

If there is a currently connected client whose session has the resourcepart being requested by the newly connecting client, the server MUST do one of the following (which of these the server does is a matter for implementation or local service policy, although suggestions are provided below).

1. Override the resourcepart provided by the newly connecting client with a server-generated resourcepart. This behavior is encouraged, because it simplifies the resource binding process for client implementations.

2. Disallow the resource binding attempt of the newly connecting client and maintain the session of the currently connected client. This behavior is neither encouraged nor discouraged, despite the fact that it was implicitly encouraged in RFC 3920; however, note that handling of the <conflict/> error is unevenly supported among existing client implementations, which often treat it as an authentication error and have been observed to discard cached credentials when receiving it.

3. Terminate the session of the currently connected client and allow the resource binding attempt of the newly connecting client. Although this was the traditional behavior of early XMPP server implementations, it is now discouraged because it can lead to a never-ending cycle of two clients effectively disconnecting each other; however, note that this behavior can be appropriate in some deployment scenarios or if the server knows that the currently connected client has a dead connection or broken stream as described under Section 4.6.

If the server follows behavior #1, it returns an <iq/> stanza of type "result" to the newly connecting client, where the <jid/> child of the <bind/> element contains XML character data that indicates the full JID of the client, including the resourcepart that was generated by the server.

```
S: <iq id='wy2xa82b4' type='result'>
     <bind xmlns='urn:ietf:params:xml:ns:xmpp-bind'>
       <jid>
   juliet@im.example.com/balcony 4db06f06-1ea4-11dc-aca3-000bcd821bfb
       </jid>
     </bind>
   </iq>
```

If the server follows behavior #2, it sends a <conflict/> stanza error (Section 8.3.3.2) in response to the resource binding attempt of the newly connecting client but maintains the XML stream so that the newly connecting client has an opportunity to negotiate a non-conflicting resourcepart (i.e., the newly connecting client needs to choose a different resourcepart before making another attempt to bind a resource).

```
S: <iq id='wy2xa82b4' type='error'>
      <error type='modify'>
        <conflict xmlns='urn:ietf:params:xml:ns:xmpp-stanzas'/>
      </error>
   </iq>
```

If the server follows behavior #3, it returns a <conflict/> stream error (Section 4.9.3.3) to the currently connected client (as described under Section 4.9.3.3) and returns an IQ stanza of type "result" (indicating success) in response to the resource binding attempt of the newly connecting client.

```
S: <iq id='wy2xa82b4' type='result'>
     <bind xmlns='urn:ietf:params:xml:ns:xmpp-bind'>
       <jid>
         juliet@im.example.com/balcony
       </jid>
     </bind>
   </iq>
```

7.7.3. Retries

If an error occurs when a client submits a resourcepart, the server SHOULD allow a configurable but reasonable number of retries (at least 5 and no more than 10); this enables the client to tolerate incorrectly provided resourceparts (e.g., bad data formats or duplicate text strings) without being forced to reconnect.

After the client has reached the retry limit, the server MUST close the stream with a <policy-violation/> stream error (Section 4.9.3.14).

8. XML Stanzas

After a client and a server (or two servers) have completed stream negotiation, either party can send XML stanzas. Three kinds of XML stanza are defined for the 'jabber:client' and 'jabber:server' namespaces: <message/>, <presence/>, and <iq/>. In addition, there are five common attributes for these stanza types. These common attributes, as well as the basic semantics of the three stanza types, are defined in this specification; more detailed information regarding the syntax of XML stanzas for instant messaging and presence applications is provided in [XMPP-IM], and for other applications in the relevant XMPP extension specifications.

Support for the XML stanza syntax and semantics defined in this specification is REQUIRED in XMPP client and server implementations.

> Security Warning: A server MUST NOT process a partial stanza and MUST NOT attach meaning to the transmission timing of any part of a stanza (before receipt of the closing tag).

8.1. Common Attributes

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

8.1.1. to

The 'to' attribute specifies the JID of the intended recipient for the stanza.

```
<message to='romeo@example.net'>
  <body>Art thou not Romeo, and a Montague?</body>
</message>
```

For information about server processing of inbound and outbound XML stanzas based on the 'to' address, refer to Section 10.

8.1.1.1. Client-to-Server Streams

The following rules apply to inclusion of the 'to' attribute in stanzas sent from a connected client to its server over an XML stream qualified by the 'jabber:client' namespace.

1. A stanza with a specific intended recipient (e.g., a conversation partner, a remote service, the server itself, even another resource associated with the user's bare JID) MUST possess a 'to' attribute whose value is an XMPP address.

2. A stanza sent from a client to a server for direct processing by the server (e.g., roster processing as described in [XMPP-IM] or presence sent to the server for broadcasting to other entities) MUST NOT possess a 'to' attribute.

The following rules apply to inclusion of the 'to' attribute in stanzas sent from a server to a connected client over an XML stream qualified by the 'jabber:client' namespace.

1. If the server has received the stanza from another connected client or from a peer server, the server MUST NOT modify the 'to' address before delivering the stanza to the client.

2. If the server has itself generated the stanza (e.g., a response to an IQ stanza of type "get" or "set", even if the stanza did not include a 'to' address), the stanza MAY include a 'to' address, which MUST be the full JID of the client; however, if the stanza does not include a 'to' address then the client MUST treat it as if the 'to' address were included with a value of the client's full JID.

Implementation Note: It is the server's responsibility to deliver only stanzas that are addressed to the client's full JID or the user's bare JID; thus, there is no need for the client to check the 'to' address of incoming stanzas. However, if the client does check the 'to' address then it is suggested to check at most the bare JID portion (not the full JID), since the 'to' address might be the user's bare JID, the client's current full JID, or even a full JID with a different resourcepart (e.g., in the case of so-called "offline messages" as described in [XEP-0160]).

8.1.1.2. Server-to-Server Streams

The following rules apply to inclusion of the 'to' attribute in the context of XML streams qualified by the 'jabber:server' namespace (i.e., server-to-server streams).

1. A stanza MUST possess a 'to' attribute whose value is an XMPP address; if a server receives a stanza that does not meet this restriction, it MUST close the stream with an <improper-addressing/> stream error (Section 4.9.3.7).

2. The domainpart of the JID contained in the stanza's 'to' attribute MUST match the FQDN of the receiving server (or any validated domain thereof) as communicated via SASL negotiation (see Section 6), Server Dialback (see [XEP-0220]), or similar means; if a server receives a stanza that does not meet this restriction, it MUST close the stream with a <host-unknown/> stream error (Section 4.9.3.6) or a <host-gone/> stream error (Section 4.9.3.5).

8.1.2. from

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

```
<message from='juliet@im.example.com/balcony'
         to='romeo@example.net'>
  <body>Art thou not Romeo, and a Montague?</body>
</message>
```

8.1.2.1. Client-to-Server Streams

The following rules apply to the 'from' attribute in the context of XML streams qualified by the 'jabber:client' namespace (i.e., client-to-server streams).

1. When a server receives an XML stanza from a connected client, the server MUST add a 'from' attribute to the stanza or override the 'from' attribute specified by the client, where the value of the

'from' attribute MUST be the full JID (<localpart@domainpart/resource>) determined by the server for the connected resource that generated the stanza (see Section 4.3.6), or the bare JID (<localpart@domainpart>) in the case of subscription-related presence stanzas (see [XMPP-IM]).

2. When the server generates a stanza on its own behalf for delivery to the client from the server itself, the stanza MUST include a 'from' attribute whose value is the bare JID (i.e., <domainpart>) of the server as agreed upon during stream negotiation (e.g., based on the 'to' attribute of the initial stream header).

3. When the server generates a stanza from the server for delivery to the client on behalf of the account of the connected client (e.g., in the context of data storage services provided by the server on behalf of the client), the stanza MUST either (a) not include a 'from' attribute or (b) include a 'from' attribute whose value is the account's bare JID (<localpart@domainpart>).

4. A server MUST NOT send to the client a stanza without a 'from' attribute if the stanza was not generated by the server on its own behalf (e.g., if it was generated by another client or a peer server and the server is merely delivering it to the client on behalf of some other entity); therefore, when a client receives a stanza that does not include a 'from' attribute, it MUST assume that the stanza is from the user's account on the server.

8.1.2.2. Server-to-Server Streams

The following rules apply to the 'from' attribute in the context of XML streams qualified by the 'jabber:server' namespace (i.e., server-to-server streams).

1. A stanza MUST possess a 'from' attribute whose value is an XMPP address; if a server receives a stanza that does not meet this restriction, it MUST close the stream with an <improper-addressing/> stream error (Section 4.9.3.7).

2. The domainpart of the JID contained in the stanza's 'from' attribute MUST match the FQDN of the sending server (or any validated domain thereof) as communicated via SASL negotiation (see Section 6), Server Dialback (see [XEP-0220]), or similar means; if a server receives a stanza that does not meet this restriction, it MUST close the stream with an <invalid-from/> stream error (Section 4.9.3.9).

Enforcement of these rules helps to prevent certain denial-of-service attacks as described under Section 13.12.

8.1.3. id

The 'id' attribute is used by the originating entity to track any response or error stanza that it might receive in relation to the generated stanza from another entity (such as an intermediate server or the intended recipient).

It is up to the originating entity whether the value of the 'id' attribute is unique only within its current stream or unique globally.

For <message/> and <presence/> stanzas, it is RECOMMENDED for the originating entity to include an 'id' attribute; for <iq/> stanzas, it is REQUIRED.

If the generated stanza includes an 'id' attribute then it is REQUIRED for the response or error stanza to also include an 'id' attribute, where the value of the 'id' attribute MUST match that of the generated stanza.

The semantics of IQ stanzas impose additional restrictions as described under Section 8.2.3.

8.1.4. type

The 'type' attribute specifies 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 stanza. The defined 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 part of the semantics for all structured request-response exchanges (no matter what the payload) and therefore are specified under Section 8.2.3. The only 'type' value common to all three kinds of stanzas is "error" as described under Section 8.3.

8.1.5. 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 [CHARSETS], "internationalization is for humans"). The value of the 'xml:lang' attribute specifies the default language of any such human-readable XML character data.

```
   <presence from='romeo@example.net/orchard' xml:lang='en'>
     <show>dnd</show>
     <status>Wooing Juliet</status>
   </presence>
```

The value of the 'xml:lang' attribute MAY be overridden by the 'xml: lang' attribute of a specific child element.

```
   <presence from='romeo@example.net/orchard' xml:lang='en'>
     <show>dnd</show>
     <status>Wooing Juliet</status>
     <status xml:lang='cs'>Dvo&#x0159;&#x00ED;m se Julii</status>
   </presence>
```

If an outbound stanza generated by a client does not possess an 'xml: lang' attribute, the client's server SHOULD add an 'xml:lang' attribute whose value is that specified for the client's output stream as defined under Section 4.7.4.

```
   C: <presence from='romeo@example.net/orchard'>
        <show>dnd</show>
        <status>Wooing Juliet</status>
      </presence>

   S: <presence from='romeo@example.net/orchard'
                to='juliet@im.example.com'
                xml:lang='en'>
        <show>dnd</show>
        <status>Wooing Juliet</status>
      </presence>
```

If an inbound stanza received by a client or server does not possess an 'xml:lang' attribute, an implementation MUST assume that the default language is that specified for the entity's input stream as defined under Section 4.7.4.

The value of the 'xml:lang' attribute MUST conform to the NMTOKEN datatype (as defined in Section 2.3 of [XML]) and MUST conform to the format defined in [LANGTAGS].

A server MUST NOT modify or delete 'xml:lang' attributes on stanzas it receives from other entities.

8.2. Basic Semantics

8.2.1. Message Semantics

The <message/> stanza is 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 will possess a 'to' attribute that specifies the intended recipient of the message (see Section 8.1.1 and Section 10.3), unless the message is being sent to the bare JID of a connected client's account. Upon receiving a message stanza with a 'to' address, a server SHOULD attempt to route or deliver it to the intended recipient (see Section 10 for general routing and delivery rules related to XML stanzas).

8.2.2. Presence Semantics

The <presence/> stanza is a specialized "broadcast" or "publish-subscribe" mechanism, whereby multiple entities receive information (in this case, network availability information) about an entity to which they have subscribed. In general, a publishing client SHOULD send a presence stanza with no 'to' attribute, in which case the server to which the client is connected will broadcast that stanza to all subscribed entities. However, a publishing client MAY also send a presence stanza with a 'to' attribute, in which case the server will route or deliver that stanza to the intended recipient. Although the <presence/> stanza is most often used by XMPP clients, it can also be used by servers, add-on services, and any other kind of XMPP entity. See Section 10 for general routing and delivery rules related to XML stanzas, and [XMPP-IM] for rules specific to presence applications.

8.2.3. IQ Semantics

Info/Query, or IQ, is a "request-response" mechanism, similar in some ways to the Hypertext Transfer Protocol [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 schema or other structural definition associated with the XML namespace that qualifies the direct child element of the IQ element (see Section 8.4), 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 can be returned in reply to a request if appropriate):

```
   Requesting                  Responding
     Entity                      Entity
   ----------                  ----------
         |                            |
         | <iq id='1' type='get'>     |
         |   [ ... payload ... ]      |
         | </iq>                      |
         | -------------------------> |
         |                            |
         | <iq id='1' type='result'>  |
         |   [ ... payload ... ]      |
         | </iq>                      |
         | <------------------------- |
         |                            |
         | <iq id='2' type='set'>     |
         |   [ ... payload ... ]      |
         | </iq>                      |
         | -------------------------> |
         |                            |
         | <iq id='2' type='error'>   |
         |   [ ... condition ... ]    |
         | </iq>                      |
         | <------------------------- |
         |                            |
```

Figure 5: Semantics of IQ Stanzas

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; if not, the recipient or an intermediate router MUST return a <bad-request/> stanza error (Section 8.3.3.1).

    * get -- The stanza requests information, inquires about what data is needed in order to complete further operations, etc.

    * set -- The stanza provides data that is needed for an operation to be completed, sets new values, replaces existing values, etc.

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

* error -- The stanza reports an error that has occurred regarding processing or delivery of a get or set request (see Section 8.3).

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 (or be empty if the generated stanza did not include an 'id' attribute).

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, the requesting entity MAY send another request (e.g., an IQ of type "set" to provide obligatory information discovered through a get/result pair).

5. An IQ stanza of type "get" or "set" MUST contain exactly one child element, which specifies the semantics of the particular request.

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

7. An IQ stanza of type "error" MAY include the child element contained in the associated "get" or "set" and MUST include an <error/> child; for details, see Section 8.3.

8.3. Stanza Errors

Stanza-related errors are handled in a manner similar to stream errors (Section 4.9). Unlike stream errors, stanza errors are recoverable; therefore, they do not result in termination of the XML stream and underlying TCP connection. Instead, the entity that discovers the error condition returns an error stanza, which is a stanza that:

o is of the same kind (message, presence, or IQ) as the generated stanza that triggered the error

o has a 'type' attribute set to a value of "error"

o typically swaps the 'from' and 'to' addresses of the generated stanza

o mirrors the 'id' attribute (if any) of the generated stanza that triggered the error

- o contains an <error/> child element that specifies the error condition and therefore provides a hint regarding actions that the sender might be able to take in an effort to remedy the error (however, it is not always possible to remedy the error)

8.3.1. Rules

The following rules apply to stanza errors:

1. The receiving or processing entity that detects an error condition in relation to a stanza SHOULD return an error stanza (and MUST do so for IQ stanzas).

2. The error stanza SHOULD simply swap the 'from' and 'to' addresses from the generated stanza, unless doing so would (1) result in an information leak (see under Section 13.10) or other breach of security, or (2) force the sender of the error stanza to include a malformed JID in the 'from' or 'to' address of the error stanza.

3. If the generated stanza was <message/> or <presence/> and included an 'id' attribute then it is REQUIRED for the error stanza to also include an 'id' attribute. If the generated stanza was <iq/> then the error stanza MUST include an 'id' attribute. In all cases, the value of the 'id' attribute MUST match that of the generated stanza (or be empty if the generated stanza did not include an 'id' attribute).

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

5. The entity that returns an error stanza MAY pass along its JID to the sender of the generated stanza (e.g., for diagnostic or tracking purposes) through the addition of a 'by' attribute to the <error/> child element.

6. The entity that returns an error stanza MAY include the original XML sent so that the sender can inspect and, if necessary, correct the XML before attempting to resend (however, this is a courtesy only and the originating entity MUST NOT depend on receiving the original payload). Naturally, the entity MUST NOT include the original data if it not well-formed XML, violates the XML restrictions of XMPP (see under Section 11.1), or is otherwise harmful (e.g., exceeds a size limit).

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

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

8.3.2. Syntax

The syntax for stanza-related errors is as follows, where XML data shown within the square brackets '[' and ']' is OPTIONAL, 'intended-recipient' is the JID of the entity to which the original stanza was addressed, 'sender' is the JID of the originating entity, and 'error-generator' is the entity that detects the fact that an error has occurred and thus returns an error stanza.

```
<stanza-kind from='intended-recipient' to='sender' type='error'>
  [OPTIONAL to include sender XML here]
  <error [by='error-generator']
         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" MUST be one of message, presence, or iq.

The "error-type" MUST be one of the following:

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

The "defined-condition" MUST correspond to one of the stanza error conditions defined under Section 8.3.3. However, because additional error conditions might be defined in the future, if an entity receives a stanza error condition that it does not understand then it MUST treat the unknown condition as equivalent to <undefined-condition/> (Section 8.3.3.21). If the designers of an XMPP protocol extension or the developers of an XMPP implementation need to communicate a stanza error condition that is not defined in this specification, they can do so by defining an application-specific error condition element qualified by an application-specific namespace.

The <error/> element:

- o MUST contain a defined condition element.
- o MAY contain a <text/> child element 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 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-specific namespace that defines the syntax and semantics of the element.

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

Interoperability Note: The syntax defined in [RFC3920] included a legacy 'code' attribute, whose semantics have been replaced by the defined condition elements; information about mapping defined condition elements to values of the legacy 'code' attribute can be found in [XEP-0086].

8.3.3. Defined Conditions

The following conditions are defined for use in stanza errors.

The error-type value that is RECOMMENDED for each defined condition is the usual expected type; however, in some circumstances a different type might be more appropriate.

8.3.3.1. bad-request

The sender has sent a stanza containing XML that does not conform to the appropriate schema or that cannot be processed (e.g., an IQ stanza that includes an unrecognized value of the 'type' attribute, or an element that is qualified by a recognized namespace but that violates the defined syntax for the element); the associated error type SHOULD be "modify".

```
   C: <iq from='juliet@im.example.com/balcony'
          id='zj3v142b'
          to='im.example.com'
          type='subscribe'>
        <ping xmlns='urn:xmpp:ping'/>
      </iq>

   S: <iq from='im.example.com'
          id='zj3v142b'
          to='juliet@im.example.com/balcony'
          type='error'>
        <error type='modify'>
          <bad-request xmlns='urn:ietf:params:xml:ns:xmpp-stanzas'/>
        </error>
      </iq>
```

8.3.3.2. conflict

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

```
   C: <iq id='wy2xa82b4' type='set'>
        <bind xmlns='urn:ietf:params:xml:ns:xmpp-bind'>
          <resource>balcony</resource>
        </bind>
      </iq>

   S: <iq id='wy2xa82b4' type='error'>
        <error type='cancel'>
          <conflict xmlns='urn:ietf:params:xml:ns:xmpp-stanzas'/>
        </error>
      </iq>
```

8.3.3.3. feature-not-implemented

The feature represented in the XML stanza is not implemented by the intended recipient or an intermediate server and therefore the stanza cannot be processed (e.g., the entity understands the namespace but does not recognize the element name); the associated error type SHOULD be "cancel" or "modify".

```
   C: <iq from='juliet@im.example.com/balcony'
          id='9u2bax16'
          to='pubsub.example.com'
          type='get'>
        <pubsub xmlns='http://jabber.org/protocol/pubsub'>
          <subscriptions/>
        </pubsub>
      </iq>

   E: <iq from='pubsub.example.com'
          id='9u2bax16'
          to='juliet@im.example.com/balcony'
          type='error'>
        <error type='cancel'>
          <feature-not-implemented
              xmlns='urn:ietf:params:xml:ns:xmpp-stanzas'/>
          <unsupported
              xmlns='http://jabber.org/protocol/pubsub#errors'
              feature='retrieve-subscriptions'/>
        </error>
      </iq>
```

8.3.3.4. forbidden

The requesting entity does not possess the necessary permissions to perform an action that only certain authorized roles or individuals are allowed to complete (i.e., it typically relates to authorization rather than authentication); the associated error type SHOULD be "auth".

```
   C: <presence
          from='juliet@im.example.com/balcony'
          id='y2bs71v4'
          to='characters@muc.example.com/JulieC'>
        <x xmlns='http://jabber.org/protocol/muc'/>
      </presence>

   E: <presence
          from='characters@muc.example.com/JulieC'
          id='y2bs71v4'
          to='juliet@im.example.com/balcony'
          type='error'>
        <error type='auth'>
          <forbidden xmlns='urn:ietf:params:xml:ns:xmpp-stanzas'/>
        </error>
      </presence>
```

8.3.3.5. gone

The recipient or server can no longer be contacted at this address, typically on a permanent basis (as opposed to the <redirect/> error condition, which is used for temporary addressing failures); the associated error type SHOULD be "cancel" and the error stanza SHOULD include a new address (if available) as the XML character data of the <gone/> element (which MUST be a Uniform Resource Identifier [URI] or Internationalized Resource Identifier [IRI] at which the entity can be contacted, typically an XMPP IRI as specified in [XMPP-URI]).

```
   C: <message
          from='juliet@im.example.com/churchyard'
          id='sj2b371v'
          to='romeo@example.net'
          type='chat'>
        <body>Thy lips are warm.</body>
      </message>

   S: <message
          from='romeo@example.net'
          id='sj2b371v'
          to='juliet@im.example.com/churchyard'
          type='error'>
        <error by='example.net'
               type='cancel'>
          <gone xmlns='urn:ietf:params:xml:ns:xmpp-stanzas'>
            xmpp:romeo@afterlife.example.net
          </gone>
        </error>
      </message>
```

8.3.3.6. internal-server-error

The server has experienced a misconfiguration or other internal error that prevents it from processing the stanza; the associated error type SHOULD be "cancel".

```
   C: <presence
          from='juliet@im.example.com/balcony'
          id='y2bs71v4'
          to='characters@muc.example.com/JulieC'>
        <x xmlns='http://jabber.org/protocol/muc'/>
      </presence>
   E: <presence
          from='characters@muc.example.com/JulieC'
          id='y2bs71v4'
          to='juliet@im.example.com/balcony'
          type='error'>
        <error type='cancel'>
          <internal-server-error
              xmlns='urn:ietf:params:xml:ns:xmpp-stanzas'/>
        </error>
      </presence>
```

8.3.3.7. item-not-found

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

```
   C: <presence from='userfoo@example.com/bar'
                id='pwb2n78i'
                to='nosuchroom@conference.example.org/foo'/>

   S: <presence from='nosuchroom@conference.example.org/foo'
                id='pwb2n78i'
                to='userfoo@example.com/bar'
                type='error'>
        <error type='cancel'>
          <item-not-found xmlns='urn:ietf:params:xml:ns:xmpp-stanzas'/>
        </error>
      </presence>
```

Security Warning: 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 (for a more detailed discussion of presence authorization, refer to the discussion of presence subscriptions in [XMPP-IM]); instead it MUST return a <service-unavailable/> stanza error (Section 8.3.3.19).

8.3.3.8. jid-malformed

The sending entity has provided (e.g., during resource binding) or communicated (e.g., in the 'to' address of a stanza) an XMPP address or aspect thereof that violates the rules defined in [XMPP-ADDR]; the associated error type SHOULD be "modify".

```
   C: <presence
          from='juliet@im.example.com/balcony'
          id='y2bs71v4'
          to='ch@r@cters@muc.example.com/JulieC'>
        <x xmlns='http://jabber.org/protocol/muc'/>
      </presence>

   E: <presence
          from='ch@r@cters@muc.example.com/JulieC'
          id='y2bs71v4'
          to='juliet@im.example.com/balcony'
          type='error'>
        <error by='muc.example.com'
               type='modify'>
          <jid-malformed
              xmlns='urn:ietf:params:xml:ns:xmpp-stanzas'/>
        </error>
      </presence>
```

Implementation Note: Enforcement of the format for XMPP localparts is primarily the responsibility of the service at which the associated account or entity is located (e.g., the example.com service is responsible for returning <jid-malformed/> errors related to all JIDs of the form <localpart@example.com>), whereas enforcement of the format for XMPP domainparts is primarily the responsibility of the service that seeks to route a stanza to the service identified by that domainpart (e.g., the example.org service is responsible for returning <jid-malformed/> errors related to stanzas that users of that service have to tried send to JIDs of the form <localpart@example.com>). However, any entity that detects a malformed JID MAY return this error.

8.3.3.9. not-acceptable

The recipient or server understands the request but cannot process it because the request does not meet criteria defined by the recipient or server (e.g., a request to subscribe to information that does not simultaneously include configuration parameters needed by the recipient); the associated error type SHOULD be "modify".

```
   C: <message to='juliet@im.example.com' id='yt2vs71m'>
        <body>[ ... the-emacs-manual ... ]</body>
      </message>
   S: <message from='juliet@im.example.com' id='yt2vs71m'>
        <error type='modify'>
          <not-acceptable
              xmlns='urn:ietf:params:xml:ns:xmpp-stanzas'/>
        </error>
      </message>
```

8.3.3.10. not-allowed

The recipient or server does not allow any entity to perform the action (e.g., sending to entities at a blacklisted domain); the associated error type SHOULD be "cancel".

```
   C: <presence
          from='juliet@im.example.com/balcony'
          id='y2bs71v4'
          to='characters@muc.example.com/JulieC'>
        <x xmlns='http://jabber.org/protocol/muc'/>
      </presence>

   E: <presence
          from='characters@muc.example.com/JulieC'
          id='y2bs71v4'
          to='juliet@im.example.com/balcony'
          type='error'>
        <error type='cancel'>
          <not-allowed xmlns='urn:ietf:params:xml:ns:xmpp-stanzas'/>
        </error>
      </presence>
```

8.3.3.11. not-authorized

The sender needs to provide credentials before being allowed to perform the action, or has provided improper credentials (the name "not-authorized", which was borrowed from the "401 Unauthorized" error of [HTTP], might lead the reader to think that this condition relates to authorization, but instead it is typically used in relation to authentication); the associated error type SHOULD be "auth".

```
   C: <presence
          from='juliet@im.example.com/balcony'
          id='y2bs71v4'
          to='characters@muc.example.com/JulieC'>
        <x xmlns='http://jabber.org/protocol/muc'/>
      </presence>
   E: <presence
          from='characters@muc.example.com/JulieC'
          id='y2bs71v4'
          to='juliet@im.example.com/balcony'>
        <error type='auth'>
          <not-authorized xmlns='urn:ietf:params:xml:ns:xmpp-stanzas'/>
        </error>
      </presence>
```

8.3.3.12. policy-violation

The entity has violated some local service policy (e.g., a message contains words that are prohibited by the service) and the server MAY choose to specify the policy in the <text/> element or in an application-specific condition element; the associated error type SHOULD be "modify" or "wait" depending on the policy being violated.

(In the following example, the client sends an XMPP message containing words that are forbidden according to the server's local service policy.)

```
   C: <message from='romeo@example.net/foo'
               to='bill@im.example.com'
               id='vq71f4nb'>
        <body>%#&@^!!!</body>
      </message>

   S: <message from='bill@im.example.com'
               id='vq71f4nb'
               to='romeo@example.net/foo'>
        <error by='example.net' type='modify'>
          <policy-violation
              xmlns='urn:ietf:params:xml:ns:xmpp-stanzas'/>
        </error>
      </message>
```

8.3.3.13. recipient-unavailable

The intended recipient is temporarily unavailable, undergoing maintenance, etc.; the associated error type SHOULD be "wait".

```
   C: <presence
          from='juliet@im.example.com/balcony'
          id='y2bs71v4'
          to='characters@muc.example.com/JulieC'>
        <x xmlns='http://jabber.org/protocol/muc'/>
      </presence>
   E: <presence
          from='characters@muc.example.com/JulieC'
          id='y2bs71v4'
          to='juliet@im.example.com/balcony'>
        <error type='wait'>
          <recipient-unavailable
              xmlns='urn:ietf:params:xml:ns:xmpp-stanzas'/>
        </error>
      </presence>
```

Security Warning: 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 (for a more detailed discussion of presence authorization, refer to the discussion of presence subscriptions in [XMPP-IM]); instead it MUST return a <service-unavailable/> stanza error (Section 8.3.3.19).

8.3.3.14. redirect

The recipient or server is redirecting requests for this information to another entity, typically in a temporary fashion (as opposed to the <gone/> error condition, which is used for permanent addressing failures); the associated error type SHOULD be "modify" and the error stanza SHOULD contain the alternate address in the XML character data of the <redirect/> element (which MUST be a URI or IRI with which the sender can communicate, typically an XMPP IRI as specified in [XMPP-URI]).

```
   C: <presence
          from='juliet@im.example.com/balcony'
          id='y2bs71v4'
          to='characters@muc.example.com/JulieC'>
        <x xmlns='http://jabber.org/protocol/muc'/>
      </presence>

   E: <presence
          from='characters@muc.example.com/JulieC'
          id='y2bs71v4'
          to='juliet@im.example.com/balcony'
          type='error'>
        <error type='modify'>
          <redirect xmlns='urn:ietf:params:xml:ns:xmpp-stanzas'>
            xmpp:characters@conference.example.org
          </redirect>
        </error>
      </presence>
```

Security Warning: An application receiving a stanza-level redirect SHOULD warn a human user of the redirection attempt and request approval before proceeding to communicate with the entity whose address is contained in the XML character data of the <redirect/> element, because that entity might have a different identity or might enforce different security policies. The end-to-end authentication or signing of XMPP stanzas could help to mitigate this risk, since it would enable the sender to determine if the entity to which it has been redirected has the same identity as the entity it originally attempted to contact. An application MAY have a policy of following redirects only if it has authenticated the receiving entity. In addition, an application SHOULD abort the communication attempt after a certain number of successive redirects (e.g., at least 2 but no more than 5).

8.3.3.15. registration-required

The requesting entity is not authorized to access the requested service because prior registration is necessary (examples of prior registration include members-only rooms in XMPP multi-user chat [XEP-0045] and gateways to non-XMPP instant messaging services, which traditionally required registration in order to use the gateway [XEP-0100]); the associated error type SHOULD be "auth".

```
   C: <presence
          from='juliet@im.example.com/balcony'
          id='y2bs71v4'
          to='characters@muc.example.com/JulieC'>
        <x xmlns='http://jabber.org/protocol/muc'/>
      </presence>

   E: <presence
          from='characters@muc.example.com/JulieC'
          id='y2bs71v4'
          to='juliet@im.example.com/balcony'>
        <error type='auth'>
          <registration-required
              xmlns='urn:ietf:params:xml:ns:xmpp-stanzas'/>
        </error>
      </presence>
```

8.3.3.16. remote-server-not-found

A remote server or service specified as part or all of the JID of the intended recipient does not exist or cannot be resolved (e.g., there is no _xmpp-server._tcp DNS SRV record, the A or AAAA fallback resolution fails, or A/AAAA lookups succeed but there is no response on the IANA-registered port 5269); the associated error type SHOULD be "cancel".

```
C: <message
       from='romeo@example.net/home'
       id='ud7n1f4h'
       to='bar@example.org'
       type='chat'>
    <body>yt?</body>
   </message>

E: <message
       from='bar@example.org'
       id='ud7n1f4h'
       to='romeo@example.net/home'
       type='error'>
     <error type='cancel'>
       <remote-server-not-found
           xmlns='urn:ietf:params:xml:ns:xmpp-stanzas'/>
     </error>
   </message>
```

8.3.3.17. remote-server-timeout

A remote server or service specified as part or all of the JID of the intended recipient (or needed to fulfill a request) was resolved but communications could not be established within a reasonable amount of time (e.g., an XML stream cannot be established at the resolved IP address and port, or an XML stream can be established but stream negotiation fails because of problems with TLS, SASL, Server Dialback, etc.); the associated error type SHOULD be "wait" (unless the error is of a more permanent nature, e.g., the remote server is found but it cannot be authenticated or it violates security policies).

```
C: <message
       from='romeo@example.net/home'
       id='ud7n1f4h'
       to='bar@example.org'
       type='chat'>
    <body>yt?</body>
   </message>
E: <message
       from='bar@example.org'
       id='ud7n1f4h'
       to='romeo@example.net/home'
       type='error'>
     <error type='wait'>
       <remote-server-timeout
           xmlns='urn:ietf:params:xml:ns:xmpp-stanzas'/>
     </error>
   </message>
```

8.3.3.18. resource-constraint

The server or recipient is busy or lacks the system resources necessary to service the request; the associated error type SHOULD be "wait".

```
   C: <iq from='romeo@example.net/foo'
          id='kj4vz31m'
          to='pubsub.example.com'
          type='get'>
        <pubsub xmlns='http://jabber.org/protocol/pubsub'>
          <items node='my_musings'/>
        </pubsub>
      </iq>

   E: <iq from='pubsub.example.com'
          id='kj4vz31m'
          to='romeo@example.net/foo'
          type='error'>
        <error type='wait'>
          <resource-constraint
              xmlns='urn:ietf:params:xml:ns:xmpp-stanzas'/>
        </error>
      </iq>
```

8.3.3.19. service-unavailable

The server or recipient does not currently provide the requested service; the associated error type SHOULD be "cancel".

```
   C: <message from='romeo@example.net/foo'
               to='juliet@im.example.com'>
        <body>Hello?</body>
      </message>
   S: <message from='juliet@im.example.com/foo'
               to='romeo@example.net'>
        <error type='cancel'>
          <service-unavailable
              xmlns='urn:ietf:params:xml:ns:xmpp-stanzas'/>
        </error>
      </message>
```

Security Warning: An application MUST return a <service-unavailable/> stanza error (Section 8.3.3.19) instead of <item-not-found/> (Section 8.3.3.7) or <recipient-unavailable/> (Section 8.3.3.13) if sending one of the latter errors would provide information about the intended recipient's network availability to an entity that is not authorized to know such information (for a more detailed discussion of presence authorization, refer to [XMPP-IM]).

8.3.3.20. subscription-required

The requesting entity is not authorized to access the requested service because a prior subscription is necessary (examples of prior subscription include authorization to receive presence information as defined in [XMPP-IM] and opt-in data feeds for XMPP publish-subscribe as defined in [XEP-0060]); the associated error type SHOULD be "auth".

```
   C: <message
          from='romeo@example.net/orchard'
          id='pa73b4n7'
          to='playwright@shakespeare.example.com'
          type='chat'>
        <subject>ACT II, SCENE II</subject>
        <body>help, I forgot my lines!</body>
      </message>

   E: <message
          from='playwright@shakespeare.example.com'
          id='pa73b4n7'
          to='romeo@example.net/orchard'
          type='error'>
        <error type='auth'>
          <subscription-required
              xmlns='urn:ietf:params:xml:ns:xmpp-stanzas'/>
        </error>
      </message>
```

8.3.3.21. undefined-condition

The error condition is not one of those defined by the other conditions in this list; any error type can be associated with this condition, and it SHOULD NOT be used except in conjunction with an application-specific condition.

```
   C: <message
          from='northumberland@shakespeare.example'
          id='richard2-4.1.247'
          to='kingrichard@royalty.england.example'>
        <body>My lord, dispatch; read o'er these articles.</body>
        <amp xmlns='http://jabber.org/protocol/amp'>
          <rule action='notify'
                condition='deliver'
                value='stored'/>
        </amp>
      </message>

   S: <message from='example.org'
               id='amp1'
               to='northumberland@example.net/field'
               type='error'>
        <amp xmlns='http://jabber.org/protocol/amp'
             from='kingrichard@example.org'
             status='error'
             to='northumberland@example.net/field'>
          <rule action='error'
                condition='deliver'
                value='stored'/>
        </amp>
        <error type='modify'>
          <undefined-condition
              xmlns='urn:ietf:params:xml:ns:xmpp-stanzas'/>
          <failed-rules xmlns='http://jabber.org/protocol/amp#errors'>
            <rule action='error'
                  condition='deliver'
                  value='stored'/>
          </failed-rules>
        </error>
      </message>
```

8.3.3.22. 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" or "modify".

```
   C: <iq from='romeo@example.net/foo'
          id='o6hsv25z'
          to='pubsub.example.com'
          type='set'>
        <pubsub xmlns='http://jabber.org/protocol/pubsub'>
           <unsubscribe
               node='my_musings'
               jid='romeo@example.net'/>
        </pubsub>
      </iq>

   E: <iq from='pubsub.example.com'
          id='o6hsv25z'
          to='romeo@example.net/foo'
          type='error'>
        <error type='modify'>
          <unexpected-request
              xmlns='urn:ietf:params:xml:ns:xmpp-stanzas'/>
          <not-subscribed
              xmlns='http://jabber.org/protocol/pubsub#errors'/>
        </error>
      </iq>
```

8.3.4. Application-Specific Conditions

As noted, an application MAY provide application-specific stanza error information by including a properly namespaced child within the error element. Typically, the application-specific element supplements or further qualifies a defined element. Thus, the <error/> element will contain two or three child elements.

```
   <iq id='ixc3v1b9' type='error'>
     <error type='modify'>
       <bad-request xmlns='urn:ietf:params:xml:ns:xmpp-stanzas'/>
       <too-many-parameters xmlns='http://example.org/ns'/>
     </error>
   </iq>
```

```
<message type='error' id='7h3baci9'>
  <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'>
      [ ... application-specific information ... ]
    </text>
    <too-many-parameters xmlns='http://example.org/ns'/>
  </error>
</message>
```

An entity that receives an application-specific error condition it does not understand MUST ignore that condition but appropriately process the rest of the error stanza.

8.4. Extended Content

Although the message, presence, and IQ stanzas provide basic semantics for messaging, availability, and request-response interactions, XMPP uses XML namespaces (see [XML-NAMES]) to extend the basic stanza syntax for the purpose of providing additional functionality.

A message or presence stanza MAY contain one or more optional child elements specifying content that extends the meaning of the message (e.g., an XHTML-formatted version of the message body as described in [XEP-0071]), and an IQ stanza of type "get" or "set" MUST contain one such child element. Such a child element MAY have any name and MUST possess a namespace declaration (other than "jabber:client", "jabber: server", or "http://etherx.jabber.org/streams") that defines the data contained within the child element. Such a child element is called an "extension element". An extension element can be included either at the direct child level of the stanza or in any mix of levels.

Similarly, "extension attributes" are allowed. That is: a stanza itself (i.e., an <iq/>, <message/>, or <presence/> element qualified by the "jabber:client" or "jabber:server" content namespace) or any child element of such a stanza (whether an extension element or a child element qualified by the content namespace) MAY also include one or more attributes qualified by XML namespaces other than the content namespace or the reserved "http://www.w3.org/XML/1998/namespace" namespace (including the so-called "empty namespace" if the attribute is not prefixed as described under [XML-NAMES]).

Interoperability Note: For the sake of backward compatibility and maximum interoperability, an entity that generates a stanza SHOULD NOT include such attributes in the stanza itself or in child elements of the stanza that are qualified by the content namespaces "jabber:client" or "jabber:server" (e.g., the <body/> child of the <message/> stanza).

An extension element or extension attribute is said to be "extended content" and the qualifying namespace for such an element or attribute is said to be an "extended namespace".

Informational Note: Although extended namespaces for XMPP are commonly defined by the XMPP Standards Foundation (XSF) and by the IETF, no specification or IETF standards action is necessary to define extended namespaces, and any individual or organization is free to define XMPP extensions.

To illustrate these concepts, several examples follow.

The following stanza contains one direct child element whose extended namespace is 'jabber:iq:roster':

```
<iq from='juliet@capulet.com/balcony'
    id='h83vxa4c'
    type='get'>
 <query xmlns='jabber:iq:roster'/>
</iq>
```

The following stanza contains two direct child elements with two different extended namespaces.

```
<presence from='juliet@capulet.com/balcony'>
  <c xmlns='http://jabber.org/protocol/caps'
     hash='sha-1'
     node='http://code.google.com/p/exodus'
     ver='QgayPKawpkPSDYmwT/WM94uAlu0='/>
  <x xmlns='vcard-temp:x:update'>
    <photo>sha1-hash-of-image</photo>
  </x>
</presence>
```

The following stanza contains two child elements, one of which is qualified by the "jabber:client" or "jabber:server" content namespace and one of which is qualified by an extended namespace; the extension element in turn contains a child element that is qualified by a different extended namespace.

```
<message to='juliet@capulet.com'>
  <body>Hello?</body>
  <html xmlns='http://jabber.org/protocol/xhtml-im'>
    <body xmlns='http://www.w3.org/1999/xhtml'>
      <p style='font-weight:bold'>Hello?</p>
    </body>
  </html>
</message>
```

It is conventional in the XMPP community for implementations to not generate namespace prefixes for elements that are qualified by extended namespaces (in the XML community, this convention is sometimes called "prefix-free canonicalization"). However, if an implementation generates such namespace prefixes then it MUST include the namespace declaration in the stanza itself or a child element of the stanza, not in the stream header (see Section 4.8.4).

Routing entities (typically servers) SHOULD try to maintain prefixes when serializing XML stanzas for processing, but receiving entities MUST NOT depend on the prefix strings to have any particular value (the allowance for the 'stream' prefix, described under Section 4.8.5, is an exception to this rule, albeit for streams rather than stanzas).

Support for any given extended namespace is OPTIONAL on the part of any implementation. If an entity does not understand such a namespace, the entity's expected behavior depends on whether the entity is (1) the recipient or (2) a server that is routing or delivering the stanza to the recipient.

If a recipient receives a stanza that contains an element or attribute it does not understand, it MUST NOT attempt to process that XML data and instead MUST proceed as follows.

- o If an intended recipient receives a message stanza whose only child element is qualified by a namespace it does not understand, then depending on the XMPP application it MUST either ignore the entire stanza or return a stanza error, which SHOULD be <service-unavailable/> (Section 8.3.3.19).

- o If an intended recipient receives a presence stanza whose only child element is qualified by a namespace it does not understand, then it MUST ignore the child element by treating the presence stanza as if it contained no child element.

- If an intended recipient receives a message or presence stanza that contains XML data qualified by a namespace it does not understand, then it MUST ignore the portion of the stanza qualified by the unknown namespace.

- If an intended recipient receives an IQ stanza of type "get" or "set" containing a child element qualified by a namespace it does not understand, then the entity MUST return an IQ stanza of type "error" with an error condition of <service-unavailable/>.

If a server handles a stanza that is intended for delivery to another entity and that contains a child element it does not understand, it MUST route the stanza unmodified to a remote server or deliver the stanza unmodified to a connected client associated with a local account.

9. Detailed Examples

The detailed examples in this section further illustrate the protocols defined in this specification.

9.1. Client-to-Server Examples

The following examples show the XMPP data flow for a client negotiating an XML stream with a server, exchanging XML stanzas, and closing the negotiated stream. The server is "im.example.com", the server requires use of TLS, the client authenticates via the SASL SCRAM-SHA-1 mechanism as <juliet@im.example.com> with a password of "r0m30myr0m30", and the client binds a client-submitted resource to the stream. It is assumed that before sending the initial stream header, the client has already resolved an SRV record of _xmpp-client._tcp.im.example.com and has opened a TCP connection to the advertised port at the resolved IP address.

9.1.1. TLS

Step 1: Client initiates stream to server:

```
C: <stream:stream
     from='juliet@im.example.com'
     to='im.example.com'
     version='1.0'
     xml:lang='en'
     xmlns='jabber:client'
     xmlns:stream='http://etherx.jabber.org/streams'>
```

Step 2: Server responds by sending a response stream header to client:

```
S: <stream:stream
     from='im.example.com'
     id='t7AMCin9zjMNwQKDnplntZPIDEI='
     to='juliet@im.example.com'
     version='1.0'
     xml:lang='en'
     xmlns='jabber:client'
     xmlns:stream='http://etherx.jabber.org/streams'>
```

Step 3: Server sends stream features to client (only the STARTTLS extension at this point, which is mandatory-to-negotiate):

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

Step 4: Client sends STARTTLS command to server:

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

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

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

Step 5 (alt): Server informs client that STARTTLS negotiation has failed, closes the XML stream, and terminates the TCP connection (thus, the stream negotiation process ends unsuccessfully and the parties do not move on to the next step):

```
S: <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 (see [TLS] for details).

Step 7: If TLS negotiation is successful, client initiates a new stream to server over the TLS-protected TCP connection:

```
C: <stream:stream
     from='juliet@im.example.com'
     to='im.example.com'
     version='1.0'
     xml:lang='en'
     xmlns='jabber:client'
     xmlns:stream='http://etherx.jabber.org/streams'>
```

Step 7 (alt): If TLS negotiation is unsuccessful, server closes TCP connection (thus, the stream negotiation process ends unsuccessfully and the parties do not move on to the next step):

9.1.2. SASL

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

```
S: <stream:stream
     from='im.example.com'
     id='vgKi/bkYME8OAj4rlXMkpucAqe4='
     to='juliet@im.example.com'
     version='1.0'
     xml:lang='en'
     xmlns='jabber:client'
     xmlns:stream='http://etherx.jabber.org/streams'>

S: <stream:features>
     <mechanisms xmlns='urn:ietf:params:xml:ns:xmpp-sasl'>
       <mechanism>SCRAM-SHA-1-PLUS</mechanism>
       <mechanism>SCRAM-SHA-1</mechanism>
       <mechanism>PLAIN</mechanism>
     </mechanisms>
   </stream:features>
```

Step 9: Client selects an authentication mechanism (in this case, SCRAM-SHA-1), including initial response data:

```
C: <auth xmlns="urn:ietf:params:xml:ns:xmpp-sasl"
         mechanism="SCRAM-SHA-1">
     biwsbj1qdWxpZXQscj1vTXNUQUF3QUFBQU1BQUFBTlAwVEFBQUFBQUJQVTBBQQ==
   </auth>
```

The decoded base 64 data is "n,,n=juliet,r=oMsTAAwAAAAMAAAANP0TAAAAAABPU0AA".

Step 10: Server sends a challenge:

```
S: <challenge xmlns="urn:ietf:params:xml:ns:xmpp-sasl">
     cj1vTXNUQUF3QUFBQU1BQUFBTlAwVEFBQUFBQUJQVBBQWUxMjQ2OTViLTY5Y
     TktNGRlNi05YzMwLWI1MWIzODA4YzU5ZSxzPU5qaGtZVE0wTURndE5HWTBaaT
     AwTmpkbUxUa3hNbVV0TkRsbU5UTm1ORE5rTURNeixpPTQwOTY=
   </challenge>
```

The decoded base 64 data is "r=oMsTAAwAAAAMAAAANP0TAAAAAABPU0AAe12469
5b-69a9-4de6-9c30-
b51b3808c59e,s=NjhkYTM0MDgtNGY0Zi00NjdmLTkxMmUtNDlmNTNmNDNkMDMz,i=409
6" (line breaks not included in actual data).

Step 11: Client sends a response:

```
C: <response xmlns="urn:ietf:params:xml:ns:xmpp-sasl">
     Yz1iaXdzLHI9b01zVEFBd0FBQUFNQUFBQU5QMFRBQUFBQUFCUFUwQUFlMTI0N
     jk1Yi02OWE5LTRkZTYtOWMzMC1iNTFiMzgwOGM1OWUscD1VQTU3dE0vU3ZwQV
     RCa0gyRlhzMFdEWHZKWXc9
   </response>
```

The decoded base 64 data is "c=biws,r=oMsTAAwAAAAMAAAANP0TAAAAAABPU0
AAe124695b-69a9-4de6-9c30-b51b3808c59e,p=UA57tM/
SvpATBkH2FXs0WDXvJYw=" (line breaks not included in actual data).

Step 12: Server informs client of success, including additional data
with success:

```
S: <success xmlns='urn:ietf:params:xml:ns:xmpp-sasl'>
     dj1wTk5ERlZFUXh1WHhDb1NFaVc4R0VaKzFSU289
   </success>
```

The decoded base 64 data is "v=pNNDFVEQxuXxCoSEiW8GEZ+1RSo=".

Step 12 (alt): Server returns a SASL error to client (thus, the
stream negotiation process ends unsuccessfully and the parties do not
move on to the next step):

```
S: <failure xmlns='urn:ietf:params:xml:ns:xmpp-sasl'>
     <not-authorized/>
   </failure>
   </stream>
```

Step 13: Client initiates a new stream to server:

```
C: <stream:stream
     from='juliet@im.example.com'
     to='im.example.com'
     version='1.0'
     xml:lang='en'
     xmlns='jabber:client'
     xmlns:stream='http://etherx.jabber.org/streams'>
```

9.1.3. Resource Binding

Step 14: Server responds by sending a stream header to client along with supported features (in this case, resource binding):

```
S: <stream:stream
     from='im.example.com'
     id='gPybzaOzBmaADgxKXu9UClbprp0='
     to='juliet@im.example.com'
     version='1.0'
     xml:lang='en'
     xmlns='jabber:client'
     xmlns:stream='http://etherx.jabber.org/streams'>

S: <stream:features>
     <bind xmlns='urn:ietf:params:xml:ns:xmpp-bind'/>
   </stream:features>
```

Upon being informed that resource binding is mandatory-to-negotiate, the client needs to bind a resource to the stream; here we assume that the client submits a human-readable text string.

Step 15: Client binds a resource:

```
C: <iq id='yhc13a95' type='set'>
     <bind xmlns='urn:ietf:params:xml:ns:xmpp-bind'>
       <resource>balcony</resource>
     </bind>
   </iq>
```

Step 16: Server accepts submitted resourcepart and informs client of successful resource binding:

```
S: <iq id='yhc13a95' type='result'>
     <bind xmlns='urn:ietf:params:xml:ns:xmpp-bind'>
       <jid>
         juliet@im.example.com/balcony
       </jid>
     </bind>
   </iq>
```

Step 16 (alt): Server returns error to client (thus, the stream negotiation process ends unsuccessfully and the parties do not move on to the next step):

```
S: <iq id='yhc13a95' type='error'>
     <error type='cancel'>
       <conflict xmlns='urn:ietf:params:xml:ns:xmpp-stanzas'/>
     </error>
   </iq>
```

9.1.4. Stanza Exchange

Now the client is allowed to send XML stanzas over the negotiated stream.

```
C: <message from='juliet@im.example.com/balcony'
            id='ju2ba41c'
            to='romeo@example.net'
            type='chat'
            xml:lang='en'>
     <body>Art thou not Romeo, and a Montague?</body>
   </message>
```

If necessary, sender's server negotiates XML streams with intended recipient's server (see Section 9.2).

The intended recipient replies, and the message is delivered to the client.

```
E: <message from='romeo@example.net/orchard'
            id='ju2ba41c'
            to='juliet@im.example.com/balcony'
            type='chat'
            xml:lang='en'>
     <body>Neither, fair saint, if either thee dislike.</body>
   </message>
```

The client can subsequently send and receive an unbounded number of subsequent XML stanzas over the stream.

9.1.5. Close

Desiring to send no further messages, the client closes its stream to the server but waits for incoming data from the server.

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

Consistent with Section 4.4, the server might send additional data to the client and then closes its stream to the client.

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

The client now sends a TLS close_notify alert, receives a responding close_notify alert from the server, and then terminates the underlying TCP connection.

9.2. Server-to-Server Examples

The following examples show the data flow for a server negotiating an XML stream with a peer server, exchanging XML stanzas, and closing the negotiated stream. The initiating server ("Server1") is im.example.com; the receiving server ("Server2") is example.net and it requires use of TLS; im.example.com presents a certificate and authenticates via the SASL EXTERNAL mechanism. It is assumed that before sending the initial stream header, Server1 has already resolved an SRV record of _xmpp-server._tcp.example.net and has opened a TCP connection to the advertised port at the resolved IP address. Note how Server1 declares the content namespace "jabber: server" as the default namespace and uses prefixes for stream-related elements, whereas Server2 uses prefix-free canonicalization.

9.2.1. TLS

Step 1: Server1 initiates stream to Server2:

```
S1: <stream:stream
      from='im.example.com'
      to='example.net'
      version='1.0'
      xmlns='jabber:server'
      xmlns:stream='http://etherx.jabber.org/streams'>
```

Step 2: Server2 responds by sending a response stream header to Server1:

```
S2: <stream
      from='example.net'
      id='hTiXkW+ih9k2SqdGkk/AZi0OJ/Q='
      to='im.example.com'
      version='1.0'
      xmlns='http://etherx.jabber.org/streams'>
```

Step 3: Server2 sends stream features to Server1 (only the STARTTLS extension at this point, which is mandatory-to-negotiate):

```
S2: <features xmlns='http://etherx.jabber.org/streams'>
      <starttls xmlns='urn:ietf:params:xml:ns:xmpp-tls'>
        <required/>
      </starttls>
    </features>
```

Step 4: Server1 sends the STARTTLS command to Server2:

```
S1: <starttls xmlns='urn:ietf:params:xml:ns:xmpp-tls'/>
```

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

```
S2: <proceed xmlns='urn:ietf:params:xml:ns:xmpp-tls'/>
```

Step 5 (alt): Server2 informs Server1 that STARTTLS negotiation has failed, closes the stream, and terminates the TCP connection (thus, the stream negotiation process ends unsuccessfully and the parties do not move on to the next step):

```
S2: <failure xmlns='urn:ietf:params:xml:ns:xmpp-tls'/>
    </stream>
```

Step 6: Server1 and Server2 attempt to complete TLS negotiation via TCP (see [TLS] for details).

Step 7: If TLS negotiation is successful, Server1 initiates a new stream to Server2 over the TLS-protected TCP connection:

```
S1: <stream:stream
      from='im.example.com'
      to='example.net'
      version='1.0'
      xmlns='jabber:server'
      xmlns:stream='http://etherx.jabber.org/streams'>
```

Step 7 (alt): If TLS negotiation is unsuccessful, Server2 closes the TCP connection (thus, the stream negotiation process ends unsuccessfully and the parties do not move on to the next step).

9.2.2. SASL

Step 8: Server2 sends a response stream header to Server1 along with available stream features (including a preference for the SASL EXTERNAL mechanism):

```
S2: <stream
      from='example.net'
      id='RChdjlgj/TIBcbT9Keu31zDihH4='
      to='im.example.com'
      version='1.0'
      xmlns='http://etherx.jabber.org/streams'>

S2: <features xmlns='http://etherx.jabber.org/streams'>
      <mechanisms xmlns='urn:ietf:params:xml:ns:xmpp-sasl'>
        <mechanism>EXTERNAL</mechanism>
      </mechanisms>
    </features>
```

Step 9: Server1 selects the EXTERNAL mechanism (including an empty response of "="):

```
S1: <auth xmlns='urn:ietf:params:xml:ns:xmpp-sasl'
          mechanism='EXTERNAL'>=</auth>
```

Step 10: Server2 returns success:

```
S2: <success xmlns='urn:ietf:params:xml:ns:xmpp-sasl'/>
```

Step 10 (alt): Server2 informs Server1 of failed authentication (thus, the stream negotiation process ends unsuccessfully and the parties do not move on to the next step):

```
S2: <failure xmlns='urn:ietf:params:xml:ns:xmpp-sasl'>
      <not-authorized/>
    </failure>
    </stream>
```

Step 11: Server1 initiates a new stream to Server2:

```
S1: <stream:stream
      from='im.example.com'
      to='example.net'
      version='1.0'
      xmlns='jabber:server'
      xmlns:stream='http://etherx.jabber.org/streams'>
```

Step 12: Server2 responds by sending a stream header to Server1 along with any additional features (or, in this case, an empty features element):

```
S2: <stream
      from='example.net'
      id='MbbV2FeojySpUIP6J91qaa+TWHM='
      to='im.example.com'
      version='1.0'
      xmlns='http://etherx.jabber.org/streams'>

S2: <features xmlns='http://etherx.jabber.org/streams'/>
```

9.2.3. Stanza Exchange

Now Server1 is allowed to send XML stanzas to Server2 over the negotiated stream from im.example.com to example.net; here we assume that the transferred stanzas are those shown earlier for client-to-server communication, albeit over a server-to-server stream qualified by the 'jabber:server' namespace.

Server1 sends XML stanza to Server2:

```
S1: <message from='juliet@im.example.com/balcony'
             id='ju2ba41c'
             to='romeo@example.net'
             type='chat'
             xml:lang='en'>
      <body>Art thou not Romeo, and a Montague?</body>
    </message>
```

9.2.4. Close

Desiring to send no further messages, Server1 closes its stream to Server2 but waits for incoming data from Server2. (In practice, the stream would most likely remain open for some time, since Server1 and Server2 do not immediately know if the stream will be needed for further communication.)

```
S1: </stream:stream>
```

Consistent with the recommended stream closing handshake, Server2 closes the stream as well:

```
S2: </stream>
```

Server1 now sends a TLS close_notify alert, receives a responding close_notify alert from Server2, and then terminates the underlying TCP connection.

10. Server Rules for Processing XML Stanzas

Each server implementation will contain its own logic for processing stanzas it receives. Such logic determines whether the server needs to route a given stanza to another domain, deliver it to a local entity (typically a connected client associated with a local account), or handle it directly within the server itself. This section provides general rules for processing XML stanzas. However, particular XMPP applications MAY specify delivery rules that modify or supplement the following rules (e.g., a set of delivery rules for instant messaging and presence applications is defined in [XMPP-IM]).

10.1. In-Order Processing

An XMPP server MUST ensure in-order processing of the stanzas and other XML elements it receives over a given input stream from a connected client or remote server.

In-order processing applies (a) to any XML elements used to negotiate and manage XML streams, and (b) to all uses of XML stanzas, including but not limited to the following:

1. Stanzas sent by a client to its server or to its own bare JID for direct processing by the server (e.g., in-order processing of a roster get and initial presence as described in [XMPP-IM]).

2. Stanzas sent by a connected client and intended for delivery to another entity associated with the server (e.g., stanzas addressed from <juliet@im.example.com> to <nurse@im.example.com>). The server MUST ensure that it delivers stanzas addressed to the intended recipient in the order it receives them over the input stream from the sending client, treating stanzas addressed to the bare JID and the full JID of the intended recipient as equivalent for delivery purposes.

3. Stanzas sent by a connected client and intended for delivery to an entity located at a remote domain (e.g., stanzas addressed from <juliet@im.example.com> to <romeo@example.net>). The routing server MUST ensure that it routes stanzas addressed to the intended recipient in the order it receives them over the input stream from the sending client, treating stanzas addressed to the bare JID and the full JID of the intended recipient as equivalent for routing purposes. To help ensure in-order processing, the routing server MUST route such stanzas over a single output stream to the remote domain, rather than sending some stanzas over one server-to-server stream and other stanzas over another server-to-server stream.

4. Stanzas routed from one server to another server for delivery to an entity associated with the remote domain (e.g., stanzas addressed from <juliet@im.example.com> to <romeo@example.net> and routed by <im.example.com> over a server-to-server stream to <example.net>). The delivering server MUST ensure that it delivers stanzas to the intended recipient in the order it receives them over the input stream from the routing server, treating stanzas addressed to the bare JID and the full JID of the intended recipient as equivalent for delivery purposes.

5. Stanzas sent by one server to another server for direct processing by the server that is hosting the remote domain (e.g., stanzas addressed from <im.example.com> to <example.net>).

If the server's processing of a particular request could have an effect on its processing of subsequent data it might receive over that input stream (e.g., enforcement of communication policies), it MUST suspend processing of subsequent data until it has processed the request.

In-order processing applies only to a single input stream. Therefore, a server is not responsible for ensuring the coherence of data it receives across multiple input streams associated with the same local account (e.g., stanzas received over two different input streams from <juliet@im.example.com/balcony> and <juliet@im.example.com/chamber>) or the same remote domain (e.g., two different input streams negotiated by a remote domain; however, a server MAY close the stream with a <conflict/> stream error (Section 4.9.3.3) if a remote server attempts to negotiate more than one stream, as described under Section 4.9.3.3).

10.2. General Considerations

At high level, there are three primary considerations at play in server processing of XML stanzas, which sometimes are at odds but need to be managed in a consistent way:

1. It is good to deliver a stanza to the intended recipient if possible.

2. If a stanza cannot be delivered, it is helpful to inform the sender.

3. It is bad to facilitate directory harvesting attacks (Section 13.11) and presence leaks (Section 13.10.2).

With regard to possible delivery-related attacks, the following points need to be kept in mind:

1. From the perspective of an attacker, there is little if any effective difference between the server's (i) delivering the stanza or storing it offline for later delivery (see [XMPP-IM]) and (ii) silently ignoring it (because an error is not returned immediately in any of those cases); therefore, in scenarios where a server delivers a stanza or places the stanza into offline storage for later delivery, it needs to silently ignore the stanza if that account does not exist.

2. How a server processes stanzas sent to the bare JID <localpart@domainpart> has implications for directory harvesting, because an attacker could determine whether an account exists if the server responds differently depending on whether there is an account for a given bare JID.

3. How a server processes stanzas sent to a full JID has implications for presence leaks, because an attacker could send requests to multiple full JIDs and receive different replies depending on whether the user has a device currently online at that full JID. The use of randomized resourceparts (whether generated by the client or the server as described under Section 7) significantly helps to mitigate this attack, so it is of somewhat lesser concern than the directory harvesting attack.

Naturally, presence is not leaked if the entity to which a user's server returns an error already knows the user's presence or is authorized to do so (e.g., by means of a presence subscription or directed presence), and a server does not enable a directory harvesting attack if it returns an error to an entity that already knows if a user exists (e.g., because the entity is in the user's contact list); these matters are discussed more fully in [XMPP-IM].

10.3. No 'to' Address

If the stanza possesses no 'to' attribute, the server MUST handle it directly on behalf of the entity that sent it, where the meaning of "handle it directly" depends on whether the stanza is message, presence, or IQ. Because all stanzas received from other servers MUST possess a 'to' attribute, this rule applies only to stanzas received from a local entity (typically a client) that is connected to the server.

10.3.1. Message

If the server receives a message stanza with no 'to' attribute, it MUST treat the message as if the 'to' address were the bare JID <localpart@domainpart> of the sending entity.

10.3.2. Presence

If the server receives a presence stanza with no 'to' attribute, it MUST broadcast it to the entities that are subscribed to the sending entity's presence, if applicable ([XMPP-IM] defines the semantics of such broadcasting for presence applications).

10.3.3. IQ

If the server receives an IQ stanza with no 'to' attribute, it MUST process the stanza on behalf of the account from which received the stanza, as follows:

1. If the IQ stanza is of type "get" or "set" and the server understands the namespace that qualifies the payload, the server MUST handle the stanza on behalf of the sending entity or return an appropriate error to the sending entity. Although the meaning of "handle" is determined by the semantics of the qualifying namespace, in general the server will respond to the IQ stanza of type "get" or "set" by returning an appropriate IQ stanza of type "result" or "error", responding as if the server were the bare JID of the sending entity. As an example, if the sending entity sends an IQ stanza of type "get" where the payload is qualified by the 'jabber:iq:roster' namespace (as described in [XMPP-IM]), then the server will return the roster associated with the sending entity's bare JID to the particular resource of the sending entity that requested the roster.

2. If the IQ stanza is of type "get" or "set" and the server does not understand the namespace that qualifies the payload, the server MUST return an error to the sending entity, which MUST be <service-unavailable/>.

3. If the IQ stanza is of type "error" or "result", the server MUST handle the error or result in accordance with the payload of the associated IQ stanza or type "get" or "set" (if there is no such associated stanza, the server MUST ignore the error or result stanza).

10.4. Remote Domain

If the domainpart of the JID contained in the 'to' attribute does not match one of the configured FQDNs of the server, the server SHOULD attempt to route the stanza to the remote domain (subject to local service provisioning and security policies regarding inter-domain communication, since such communication is OPTIONAL for any given deployment). As described in the following sections, there are two possible cases.

> Security Warning: These rules apply only client-to-server streams. As described under Section 8.1.1.2, a server MUST NOT accept a stanza over a server-to-server stream if the domainpart of the JID in the 'to' attribute does not match an FQDN serviced by the receiving server.

10.4.1. Existing Stream

If a server-to-server stream already exists between the two domains, the sender's server SHOULD attempt to route the stanza to the authoritative server for the remote domain over the existing stream.

10.4.2. No Existing Stream

If there exists no server-to-server stream between the two domains, the sender's server will proceed as follows:

1. Resolve the FQDN of the remote domain (as described under Section 13.9.2).

2. Negotiate a server-to-server stream between the two domains (as defined under Section 5 and Section 6).

3. Route the stanza to the authoritative server for the remote domain over the newly established stream.

10.4.3. Error Handling

If routing of a stanza to the intended recipient's server is unsuccessful, the sender's server MUST return an error to the sender. If resolution of the remote domain is unsuccessful, the stanza error MUST be <remote-server-not-found/> (Section 8.3.3.16). If resolution succeeds but streams cannot be negotiated, the stanza error MUST be <remote-server-timeout/> (Section 8.3.3.17).

If stream negotiation with the intended recipient's server is successful but the remote server cannot deliver the stanza to the recipient, the remote server MUST return an appropriate error to the sender by way of the sender's server.

10.5. Local Domain

If the domainpart of the JID contained in the 'to' attribute matches one of the configured FQDNs of the server, the server MUST first determine if the FQDN is serviced by the server itself or by a specialized local service. If the latter, the server MUST route the stanza to that service. If the former, the server MUST proceed as follows. However, the server MUST NOT route or "forward" the stanza to another domain, because it is the server's responsibility to process all stanzas for which the domainpart of the 'to' address matches one of the configured FQDNs of the server (among other things, this helps to prevent looping).

10.5.1. domainpart

If the JID contained in the 'to' attribute is of the form <domainpart>, then the server MUST either (a) handle the stanza as appropriate for the stanza kind or (b) return an error stanza to the sender.

10.5.2. domainpart/resourcepart

If the JID contained in the 'to' attribute is of the form <domainpart/resourcepart>, then the server MUST either (a) handle the stanza as appropriate for the stanza kind or (b) return an error stanza to the sender.

10.5.3. localpart@domainpart

An address of this type is normally associated with an account on the server. The following rules provide some general guidelines; more detailed rules in the context of instant messaging and presence applications are provided in [XMPP-IM].

10.5.3.1. No Such User

If there is no local account associated with the <localpart@domainpart>, how the stanza is processed depends on the stanza type.

- For a message stanza, the server MUST either (a) silently ignore the stanza or (b) return a <service-unavailable/> stanza error (Section 8.3.3.19) to the sender.

- For a presence stanza, the server SHOULD ignore the stanza (or behave as described in [XMPP-IM]).

- For an IQ stanza, the server MUST return a <service-unavailable/> stanza error (Section 8.3.3.19) to the sender.

10.5.3.2. User Exists

If the JID contained in the 'to' attribute is of the form <localpart@domainpart>, how the stanza is processed depends on the stanza type.

- For a message stanza, if there exists at least one connected resource for the account then the server SHOULD deliver it to at least one of the connected resources. If there exists no connected resource then the server MUST either (a) store the message offline for delivery when the account next has a connected resource or (b) return a <service-unavailable/> stanza error (Section 8.3.3.19).

- For a presence stanza, if there exists at least one connected resource that has sent initial presence (i.e., has a "presence session" as defined in [XMPP-IM]) then the server SHOULD deliver it to such resources. If there exists no connected resource then the server SHOULD ignore the stanza (or behave as described in [XMPP-IM]).

- For an IQ stanza, the server MUST handle it directly on behalf of the intended recipient.

10.5.4. localpart@domainpart/resourcepart

If the JID contained in the 'to' attribute is of the form <localpart@domainpart/resourcepart> and the user exists but there is no connected resource that exactly matches the full JID, the stanza SHOULD be processed as if the JID were of the form <localpart@domainpart> as described under Section 10.5.3.2.

If the JID contained in the 'to' attribute is of the form <localpart@domainpart/resourcepart>, the user exists, and there is a connected resource that exactly matches the full JID, the server MUST deliver the stanza to that connected resource.

11. XML Usage

11.1. XML Restrictions

XMPP defines a class of data objects called XML streams as well as the behavior of computer programs that process XML streams. XMPP is an application profile or restricted form of the Extensible Markup Language [XML], and a complete XML stream (including start and end stream tags) is a conforming XML document.

However, XMPP does not deal with XML documents but with XML streams. 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]. Furthermore, XMPP uses XML to define protocol data structures and extensions for the purpose of structured interactions between network entities and therefore adheres to the recommendations provided in [XML-GUIDE] regarding restrictions on the use of XML in IETF protocols. As a result, the following features of XML are prohibited in XMPP:

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

An XMPP implementation MUST behave as follows with regard to these features:

1. An XMPP implementation MUST NOT inject characters matching such features into an XML stream.
2. If an XMPP implementation receives characters matching such features over an XML stream, it MUST close the stream with a stream error, which SHOULD be <restricted-xml/> (Section 4.9.3.18), although some existing implementations send <bad-format/> (Section 4.9.3.1) instead.

11.2. XML Namespace Names and Prefixes

XML namespaces (see [XML-NAMES]) are used within XMPP streams 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 streams are namespace-aware enables any allowable XML to be structurally mixed with any data element within XMPP. XMPP-specific rules for XML namespace names and prefixes are defined under Section 4.8 for XML streams and Section 8.4 for XML stanzas.

11.3. Well-Formedness

In XML, there are two varieties of well-formedness:

- "XML-well-formedness" in accordance with the definition of "well-formed" from Section 2.1 of [XML].
- "Namespace-well-formedness" in accordance with the definition of "namespace-well-formed" from Section 7 of [XML-NAMES].

The following rules apply:

1. An XMPP entity MUST NOT generate data that is not XML-well-formed.
2. An XMPP entity MUST NOT accept data that is not XML-well-formed; instead it MUST close the stream over which the data was received with a <not-well-formed/> stream error (Section 4.9.3.13).
3. An XMPP entity MUST NOT generate data that is not namespace-well-formed.
4. An XMPP entity MUST NOT accept data that is not namespace-well-formed (in particular, an XMPP server MUST NOT route or deliver data that is not namespace-well-formed); instead it MUST return either a <not-acceptable/> stanza error (Section 8.3.3.9) or close the stream with a <not-well-formed/> stream error (Section 4.9.3.13), where it is preferable to close the stream with a stream error because accepting such data can open an entity to certain denial-of-service attacks.

Interoperability Note: Because these restrictions were underspecified in [RFC3920], it is possible that implementations based on that specification will send data that does not comply with these restrictions.

11.4. Validation

A server is not responsible for ensuring that XML data delivered to a connected client or routed to a peer server is valid, in accordance with the definition of "valid" provided in Section 2.8 of [XML]. An implementation MAY choose to accept or send only data that has been explicitly validated against the schemas provided in this document, but such behavior is OPTIONAL. Clients are advised not to rely on the ability to send data that does not conform to the schemas, and SHOULD ignore any non-conformant elements or attributes on the incoming XML stream.

Informational Note: The terms "valid" and "well-formed" are distinct in XML.

11.5. Inclusion of XML Declaration

Before sending a stream header, an implementation SHOULD send an XML declaration (matching the "XMLDecl" production from [XML]). Applications MUST follow the rules provided in [XML] regarding the format of the XML declaration and the circumstances under which the XML declaration is included.

Because external markup declarations are prohibited in XMPP (as described under Section 11.1), the standalone document declaration (matching the "SDDecl" production from [XML]) would have no meaning and therefore MUST NOT be included in an XML declaration sent over an XML stream. If an XMPP entity receives an XML declaration containing a standalone document declaration set to a value of "no", the entity MUST either ignore the standalone document declaration or close the stream with a stream error, which SHOULD be <restricted-xml/> (Section 4.9.3.18).

11.6. Character Encoding

Implementations MUST support the UTF-8 transformation of Universal Character Set [UCS2] characters, as needed for conformance with [CHARSETS] and as defined in [UTF-8]. Implementations MUST NOT attempt to use any other encoding. If one party to an XML stream detects that the other party has attempted to send XML data with an encoding other than UTF-8, it MUST close the stream with a stream error, which SHOULD be <unsupported-encoding/> (Section 4.9.3.22), although some existing implementations send <bad-format/> (Section 4.9.3.1) instead.

Because it is mandatory for an XMPP implementation to support all and only the UTF-8 encoding and because UTF-8 always has the same byte order, an implementation MUST NOT send a byte order mark ("BOM") at the beginning of the data stream. If an entity receives the [UNICODE] character U+FEFF anywhere in an XML stream (including as the first character of the stream), it MUST interpret that character as a zero width no-break space, not as a byte order mark.

11.7. Whitespace

Except where explicitly disallowed (e.g., during TLS negotiation (Section 5) and SASL negotiation (Section 6)), either entity MAY send whitespace as separators between XML stanzas or between any other first-level elements sent over the stream. One common use for sending such whitespace is explained under Section 4.4.

11.8. XML Versions

XMPP is an application profile of XML 1.0. A future version of XMPP might be defined in terms of higher versions of XML, but this specification defines XMPP only in terms of XML 1.0.

12. Internationalization Considerations

As specified under Section 11.6, XML streams MUST be encoded in UTF-8.

As specified under Section 4.7, an XML stream SHOULD include an 'xml: lang' attribute specifying the default language for any XML character data that is intended to be presented to a human user. As specified under Section 8.1.5, 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 on stanzas it receives from other entities.

Internationalization of XMPP addresses is specified in [XMPP-ADDR].

13. Security Considerations

13.1. Fundamentals

XMPP technologies are typically deployed using a decentralized client-server architecture. As a result, several paths are possible when two XMPP entities need to communicate:

1. Both entities are servers. In this case, the entities can establish a direct server-to-server stream between themselves.

2. One entity is a server and the other entity is a client whose account is hosted on that server. In this case, the entities can establish a direct client-to-server stream between themselves.

3. Both entities are clients whose accounts are hosted on the same server. In this case, the entities cannot establish a direct stream between themselves, but there is only one intermediate entity between them, whose policies they might understand and in which they might have some level of trust (e.g., the server might require the use of Transport Layer Security for all client connections).

4. Both entities are clients but their accounts are hosted on different servers. In this case, the entities cannot establish a direct stream between themselves and there are two intermediate entities between them; each client might have some trust in the server that hosts its account but might know nothing about the policies of the server to which the other client connects.

This specification covers only the security of a direct XML stream between two servers or between a client and a server (cases #1 and #2), where each stream can be considered a single "hop" along a communication path. The goal of security for a multi-hop path (cases #3 and #4), although very desirable, is out of scope for this specification.

In accordance with [SEC-GUIDE], this specification covers communication security (confidentiality, data integrity, and peer entity authentication), non-repudiation, and systems security (unauthorized usage, inappropriate usage, and denial of service). We also discuss common security issues such as information leaks, firewalls, and directory harvesting, as well as best practices related to the reuse of technologies such as base 64, DNS, cryptographic hash functions, SASL, TLS, UTF-8, and XML.

13.2. Threat Model

The threat model for XMPP is in essence the standard "Internet Threat Model" described in [SEC-GUIDE]. Attackers are assumed to be interested in and capable of launching the following attacks against unprotected XMPP systems:

- Eavesdropping
- Sniffing passwords
- Breaking passwords through dictionary attacks

- Discovering usernames through directory harvesting attacks
- Replaying, inserting, deleting, or modifying stanzas
- Spoofing users
- Gaining unauthorized entry to a server or account
- Using a server or account inappropriately
- Denying service to other entities
- Subverting communication streams through man-in-the-middle attacks
- Gaining control over on-path servers

Where appropriate, the following sections describe methods for protecting against these threats.

13.3. Order of Layers

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

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

This order has important security implications, as described throughout these security considerations.

Within XMPP, XML stanzas are further ordered on top of XML streams, as described under Section 4.

13.4. Confidentiality and Integrity

The use of Transport Layer Security (TLS) with appropriate ciphersuites provides a reliable mechanism to ensure the confidentiality and integrity of data exchanged between a client and a server or between two servers. Therefore, TLS can help to protect against eavesdropping, password sniffing, man-in-the-middle attacks, and stanza replays, insertion, deletion, and modification over an XML stream. XMPP clients and servers MUST support TLS as defined under Section 5.

Informational Note: The confidentiality and integrity of a stream can be protected by methods other than TLS, e.g., by means of a SASL mechanism that involves negotiation of a security layer.

Security Warning: The use of TLS in XMPP applies to a single stream. Because XMPP is typically deployed using a distributed client-server architecture (as explained under Section 2.5), a stanza might traverse multiple streams, and not all of those streams might be TLS-protected. For example, a stanza sent from a client with a session at one server (e.g., <romeo@example.net/orchard>) and intended for delivery to a client with a session at another server (e.g., <juliet@example.com/balcony>) will traverse three streams: (1) the stream from the sender's client to its server, (2) the stream from the sender's server to the recipient's server, and (3) the stream from the recipient's server to the recipient's client. Furthermore, the stanza will be processed as cleartext within the sender's server and the recipient's server. Therefore, even if the stream from the sender's client to its server is protected, the confidentiality and integrity of a stanza sent over that protected stream cannot be guaranteed when the stanza is processed by the sender's server, sent from the sender's server to the recipient's server, processed by the recipient's server, or sent from the recipient's server to the recipient's client. Only a robust technology for end-to-end encryption could ensure the confidentiality and integrity of a stanza as it traverses all of the "hops" along a communication path (e.g., a technology that meets the requirements defined in [E2E-REQS]). Unfortunately, the XMPP community has so far failed to produce an end-to-end encryption technology that might be suitable for widespread implementation and deployment, and definition of such a technology is out of scope for this document.

13.5. Peer Entity Authentication

The use of the Simple Authentication and Security Layer (SASL) for authentication provides a reliable mechanism for peer entity authentication. Therefore, SASL helps to protect against user spoofing, unauthorized usage, and man-in-the middle attacks. XMPP clients and servers MUST support SASL as defined under Section 6.

13.6. Strong Security

[STRONGSEC] defines "strong security" and its importance to communication over the Internet. For the purpose of XMPP communication over client-to-server and server-to-server streams, the term "strong security" refers to the use of security technologies that provide both mutual authentication and integrity checking (e.g., a combination of TLS encryption and SASL authentication using appropriate SASL mechanisms).

Implementations MUST support strong security. Service provisioning SHOULD use strong security.

An implementation SHOULD make it possible for an end user or service administrator to provision a deployment with specific trust anchors for the certificate presented by a connecting entity (either client or server); when an application is thus provisioned, it MUST NOT use a generic PKI trust store to authenticate the connecting entity. More detailed rules and guidelines regarding certificate validation are provided in the next section.

The initial stream and the response stream MUST be secured separately, although security in both directions MAY be established via mechanisms that provide mutual authentication.

13.7. Certificates

Channel encryption of an XML stream using Transport Layer Security as described under Section 5, and in some cases also authentication as described under Section 6, is commonly based on a PKIX certificate presented by the receiving entity (or, in the case of mutual certificate authentication, both the receiving entity and the initiating entity). This section describes best practices regarding the generation of PKIX certificates to be presented by XMPP entities and the verification of PKIX certificates presented by XMPP entities.

In general, the following sections rely on and extend the rules and guidelines provided in the [PKIX] profile of [X509], and in [TLS-CERTS]. The reader is referred to those specifications for a detailed understanding of PKIX certificates and their use in TLS.

13.7.1. Certificate Generation

13.7.1.1. General Considerations

The following rules apply to end entity public key certificates that are issued to XMPP servers or clients:

1. The certificate MUST conform to [PKIX].

2. The certificate MUST NOT contain a basicConstraints extension with the cA boolean set to TRUE.

3. The subject field MUST NOT be null.

4. The signatureAlgorithm SHOULD be the PKCS #1 version 1.5 signature algorithm with SHA-256 as defined by [PKIX-ALGO], or a stronger algorithm if available.

5. The certificate SHOULD include an Authority Information Access (AIA) extension that specifies the address of an Online Certificate Status Protocol [OCSP] responder (if not, a relying party would need to fall back on the use of Certificate Revocation Lists (CRLs) as described in [PKIX]).

The following rules apply to certification authority (CA) certificates that are used by issuers of XMPP end entity certificates:

1. The certificate MUST conform to [PKIX].

2. The certificate MUST contain a keyUsage extension with the digitalSignature bit set.

3. The subject field MUST NOT be null.

4. The signatureAlgorithm SHOULD be the PKCS #1 version 1.5 signature algorithm with SHA-256 as defined by [PKIX-ALGO], or a stronger algorithm if available.

5. For issuers of public key certificates, the issuer's certificate MUST contain a basicConstraints extension with the cA boolean set to TRUE.

13.7.1.2. Server Certificates

13.7.1.2.1. Rules

In a PKIX certificate to be presented by an XMPP server (i.e., a "server certificate"), the certificate SHOULD include one or more XMPP addresses (i.e., domainparts) associated with XMPP services hosted at the server. The rules and guidelines defined in [TLS-CERTS] apply to XMPP server certificates, with the following XMPP-specific considerations:

o Support for the DNS-ID identifier type [PKIX] is REQUIRED in XMPP client and server software implementations. Certification authorities that issue XMPP-specific certificates MUST support the DNS-ID identifier type. XMPP service providers SHOULD include the DNS-ID identifier type in certificate requests.

- Support for the SRV-ID identifier type [PKIX-SRV] is REQUIRED for XMPP client and server software implementations (for verification purposes XMPP client implementations need to support only the "_xmpp-client" service type, whereas XMPP server implementations need to support both the "_xmpp-client" and "_xmpp-server" service types). Certification authorities that issue XMPP-specific certificates SHOULD support the SRV-ID identifier type. XMPP service providers SHOULD include the SRV-ID identifier type in certificate requests.

- Support for the XmppAddr identifier type (specified under Section 13.7.1.4) is encouraged in XMPP client and server software implementations for the sake of backward-compatibility, but is no longer encouraged in certificates issued by certification authorities or requested by XMPP service providers.

- DNS domain names in server certificates MAY contain the wildcard character '*' as the complete left-most label within the identifier.

13.7.1.2.2. Examples

For our first (relatively simple) example, consider a company called "Example Products, Inc." It hosts an XMPP service at "im.example.com" (i.e., user addresses at the service are of the form "user@im.example.com"), and SRV lookups for the xmpp-client and xmpp-server services at "im.example.com" yield one machine, called "x.example.com", as follows:

```
_xmpp-client._tcp.im.example.com. 400 IN SRV 20 0 5222 x.example.com
_xmpp-server._tcp.im.example.com. 400 IN SRV 20 0 5269 x.example.com
```

The certificate presented by x.example.com contains the following representations:

- An otherName type of SRVName (id-on-dnsSRV) containing an IA5String (ASCII) string of "_xmpp-client.im.example.com"

- An otherName type of SRVName (id-on-dnsSRV) containing an IA5String (ASCII) string of "_xmpp-server.im.example.com"

- A dNSName containing an ASCII string of "im.example.com"

- An otherName type of XmppAddr (id-on-xmppAddr) containing a UTF-8 string of "im.example.com"

- A CN containing an ASCII string of "Example Products, Inc."

For our second (more complex) example, consider an ISP called "Example Internet Services". It hosts an XMPP service at "example.net" (i.e., user addresses at the service are of the form "user@example.net"), but SRV lookups for the xmpp-client and xmpp-server services at "example.net" yield two machines ("x1.example.net" and "x2.example.net"), as follows:

```
_xmpp-client._tcp.example.net. 68400 IN SRV 20 0 5222 x1.example.net.
_xmpp-client._tcp.example.net. 68400 IN SRV 20 0 5222 x2.example.net.
_xmpp-server._tcp.example.net. 68400 IN SRV 20 0 5269 x1.example.net.
_xmpp-server._tcp.example.net. 68400 IN SRV 20 0 5269 x2.example.net.
```

Example Internet Services also hosts chatrooms at chat.example.net, and provides an xmpp-server SRV record for that service as well (thus enabling entities from remote domains to access that service). It also might provide other such services in the future, so it wishes to represent a wildcard in its certificate to handle such growth.

The certificate presented by either x1.example.net or x2.example.net contains the following representations:

- An otherName type of SRVName (id-on-dnsSRV) containing an IA5String (ASCII) string of "_xmpp-client.example.net"

- An otherName type of SRVName (id-on-dnsSRV) containing an IA5String (ASCII) string of "_xmpp-server.example.net"

- An otherName type of SRVName (id-on-dnsSRV) containing an IA5String (ASCII) string of "_xmpp-server.chat.example.net"

- A dNSName containing an ASCII string of "example.net"

- A dNSName containing an ASCII string of "*.example.net"

- An otherName type of XmppAddr (id-on-xmppAddr) containing a UTF-8 string of "example.net"

- An otherName type of XmppAddr (id-on-xmppAddr) containing a UTF-8 string of "chat.example.net"

- A CN containing an ASCII string of "Example Internet Services"

13.7.1.3. Client Certificates

In a PKIX certificate to be presented by an XMPP client controlled by a human user (i.e., a "client certificate"), it is RECOMMENDED for the certificate to include one or more JIDs associated with an XMPP user. If included, a JID MUST be represented as an XmppAddr as specified under Section 13.7.1.4.

13.7.1.4. XmppAddr Identifier Type

The XmppAddr identifier type is a UTF8String within an otherName entity inside the subjectAltName, using the [ASN.1] Object Identifier "id-on-xmppAddr" specified below.

```
   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
```

As an alternative to the "id-on-xmppAddr" notation, this Object Identifier MAY be represented in dotted display format (i.e., "1.3.6.1.5.5.7.8.5") or in the Uniform Resource Name notation specified in [URN-OID] (i.e., "urn:oid:1.3.6.1.5.5.7.8.5").

Thus for example the JID <juliet@im.example.com> as included in a certificate could be formatted in any of the following three ways:

```
   id-on-xmppAddr:
      subjectAltName=otherName:id-on-xmppAddr;UTF8:juliet@im.example.com

   dotted display format:  subjectAltName=otherName:
      1.3.6.1.5.5.7.8.5;UTF8:juliet@im.example.com

   URN notation:  subjectAltName=otherName:urn:oid:
      1.3.6.1.5.5.7.8.5;UTF8:juliet@im.example.com
```

Use of the "id-on-xmppAddr" format is RECOMMENDED in the generation of certificates, but all three formats MUST be supported for the purpose of certificate validation.

The "id-on-xmppAddr" object identifier MAY be used in conjunction with the extended key usage extension specified in Section 4.2.1.12 of [PKIX] in order to explicitly define and limit the intended use of a certificate to the XMPP network.

13.7.2. Certificate Validation

When an XMPP entity is presented with a server certificate or client certificate by a peer for the purpose of encryption or authentication of XML streams as described under Section 5 and Section 6, the entity MUST attempt to validate the certificate to determine if the certificate will be considered a "trusted certificate", i.e., a certificate that is acceptable for encryption and/or authentication in accordance with the XMPP entity's local service policies or configured settings.

For both server certificates and client certificates, the validating entity MUST do the following:

1. Attempt to verify the integrity of the certificate.

2. Attempt to verify that the certificate has been properly signed by the issuing Certificate Authority.

3. Attempt to validate the full certification path.

4. Check the rules for end entity public key certificates and certification authority certificates specified under Section 13.7.1.1 for the general case and under either Section 13.7.1.2 or Section 13.7.1.3 for XMPP server or client certificates, respectively.

5. Check certificate revocation messages via Certificate Revocation Lists (CRLs), the Online Certificate Status Protocol [OCSP], or both.

If any of those validation attempts fail, the validating entity MUST unilaterally terminate the session.

The following sections describe the additional identity verification rules that apply to server-to-server and client-to-server streams.

Once the identity of the stream peer has been validated, the validating entity SHOULD also correlate the validated identity with the 'from' address (if any) of the stream header it received from the peer. If the two identities do not match, the validating entity SHOULD terminate the connection attempt (however, there might be good reasons why the identities do not match, as described under Section 4.7.1).

13.7.2.1. Server Certificates

For server certificates, the rules and guidelines defined in [TLS-CERTS] apply, with the proviso that the XmppAddr identifier specified under Section 13.7.1.4 is allowed as a reference identifier.

The identities to be checked are set as follows:

- The initiating entity sets the source domain of its reference identifiers to the 'to' address it communicates in the initial stream header; i.e., this is the identity it expects the receiving entity to provide in a PKIX certificate.

- The receiving entity sets the source domain of its reference identifiers to the 'from' address communicated by the initiating entity in the initial stream header; i.e., this is the identity that the initiating entity is trying to assert.

In the case of server-to-server communication, the matching procedure described in [TLS-CERTS] can be performed by an application server (receiving entity) when verifying an incoming server-to-server connection from a peer server (initiating entity). In this case, the receiving entity verifies the identity of the initiating entity and uses as the source domain of its reference identifiers the DNS domain name asserted by the initiating entity in the 'from' attribute of the initial stream header. However, the matching procedure described in [TLS-CERTS] remains unchanged and is applied in the same way.

13.7.2.2. Client Certificates

When an XMPP server validates a certificate presented by a client, there are three possible cases, as discussed in the following sections.

The identities to be checked are set as follows:

- The client sets the source domain of its reference identifiers to the 'to' address it communicates in the initial stream header; i.e., this is the identity it expects the server to provide in a PKIX certificate.

- The server sets the bare JID of its reference identifiers to the 'from' address communicated by the initiating entity in the initial stream header; i.e., this is the identity that the client is trying to assert.

13.7.2.2.1. Case #1

If the client certificate appears to be certified by a certification path terminating in a trust anchor (as described in Section 6.1 of [PKIX]), the server MUST check the certificate for any instances of the XmppAddr as described under Section 13.7.1.4. There are three possible sub-cases:

Sub-Case #1: The server finds one XmppAddr for which the domainpart of the represented JID matches one of the configured FQDNs of the server; the server SHOULD use this represented JID as the validated identity of the client.

Sub-Case #2: The server finds more than one XmppAddr for which the domainpart of the represented JID matches one of the configured FQDNs of the server; the server SHOULD use one of these represented JIDs as the validated identity of the client, choosing among them based on the bare JID contained in the 'from' address of the initial stream header (if any), based on the domainpart contained in the 'to' address of the initial stream header, or in accordance with local service policies (such as a lookup in a user database based on other information contained in the client certificate).

Sub-Case #3: The server finds no XmppAddrs, or finds at least one XmppAddr but the domainpart of the represented JID does not match one of the configured FQDNs of the server; the server MUST NOT use the represented JID (if any) as the validated identity of the client but instead MUST validate the identity of the client using other means in accordance with local service policies (such as a lookup in a user database based on other information contained in the client certificate). If the identity cannot be so validated, the server MAY abort the validation process and terminate the TLS negotiation.

13.7.2.2.2. Case #2

If the client certificate is certified by a Certificate Authority not known to the server, the server MUST proceed as under Case #1, Sub-Case #3.

13.7.2.2.3. Case #3

If the client certificate is self-signed, the server MUST proceed as under Case #1, Sub-Case #3.

13.7.2.3. Checking of Certificates in Long-Lived Streams

Because XMPP uses long-lived XML streams, it is possible that a certificate presented during stream negotiation might expire or be revoked while the stream is still live (this is especially relevant in the context of server-to-server streams). Therefore, each party to a long-lived stream SHOULD:

1. Cache the expiration date of the certificate presented by the other party and any certificates on which that certificate depends (such as a root or intermediate certificate for a certification authority), and close the stream when any such certificate expires, with a stream error of <reset/> (Section 4.9.3.16).

2. Periodically query the Online Certificate Status Protocol [OCSP] responder listed in the Authority Information Access (AIA) extension of the certificate presented by the other party and any certificates on which that certificate depends (such as a root or intermediate certificate for a certification authority), and close the stream if any such certificate has been revoked, with a stream error of <reset/> (Section 4.9.3.16). It is RECOMMENDED to query the OSCP responder at or near the time communicated via the nextUpdate field received in the OCSP response or, if the nextUpdate field is not set, to query every 24 hours.

After the stream is closed, the initiating entity from the closed stream will need to reconnect and the receiving entity will need to authenticate the initiating entity based on whatever certificate it presents during negotiation of the new stream.

13.7.2.4. Use of Certificates in XMPP Extensions

Certificates MAY be used in extensions to XMPP for the purpose of application-layer encryption or authentication above the level of XML streams (e.g., for end-to-end encryption). Such extensions will define their own certificate handling rules. At a minimum, such extensions are encouraged to remain consistent with the rules defined in this specification, specifying additional rules only when necessary.

13.8. Mandatory-to-Implement TLS and SASL Technologies

The following TLS ciphersuites and SASL mechanisms are mandatory-to-implement (naturally, implementations MAY support other ciphersuites and mechanisms as well). For security considerations related to TLS ciphersuites, see Section 13.9.4 and [TLS]. For security considerations related to SASL mechanisms, see Section 13.9.4, [SASL], and specifications for particular SASL mechanisms such as [SCRAM], [DIGEST-MD5], and [PLAIN].

13.8.1. For Authentication Only

For authentication only, servers and clients MUST support the SASL Salted Challenge Response Authentication Mechanism [SCRAM] -- in particular, the SCRAM-SHA-1 and SCRAM-SHA-1-PLUS variants.

> Security Warning: Even though it is possible to complete authentication only without confidentiality, it is RECOMMENDED for servers and clients to protect the stream with TLS before attempting authentication with SASL, both to help protect the information exchanged during SASL negotiation and to help prevent certain downgrade attacks as described under Section 13.9.4 and Section 13.9.5. Even if TLS is used, implementations SHOULD also enforce channel binding as described under Section 13.9.4.

> Interoperability Note: The SCRAM-SHA-1 or SASL-SCRAM-SHA-1-PLUS variants of the SCRAM mechanism replace the SASL DIGEST-MD5 mechanism as XMPP's mandatory-to-implement password-based method for authentication only. For backward-compatibility with existing deployed infrastructure, implementations are encouraged to continue supporting the DIGEST-MD5 mechanism as specified in [DIGEST-MD5]; however, there are known interoperability issues with DIGEST-MD5 that make it impractical in the long term.

13.8.2. For Confidentiality Only

For confidentiality only, servers MUST support TLS with the TLS_RSA_WITH_AES_128_CBC_SHA ciphersuite.

> Security Warning: Because a connection with confidentiality only has weaker security properties than a connection with both confidentiality and authentication, it is RECOMMENDED for servers and clients to prefer connections with both qualities (e.g., by protecting the stream with TLS before attempting authentication with SASL). In practice, confidentiality only is employed merely for server-to-server connections when the peer server does not present a trusted certificate and the servers use Server Dialback [XEP-0220] for weak identity verification, but TLS with confidentiality only is still desirable to protect the connection against casual eavesdropping.

13.8.3. For Confidentiality and Authentication with Passwords

For both confidentiality and authentication with passwords, servers and clients MUST implement TLS with the TLS_RSA_WITH_AES_128_CBC_SHA ciphersuite plus SASL SCRAM, in particular the SCRAM-SHA-1 and SCRAM-SHA-1-PLUS variants (with SCRAM-SHA1-PLUS being preferred, as described under Section 13.9.4).

As further explained in the following Security Warning, in certain circumstances a server MAY offer TLS with the TLS_RSA_WITH_AES_128_CBC_SHA ciphersuite plus SASL PLAIN when it is not possible to offer more secure alternatives; in addition, clients SHOULD implement PLAIN over TLS in order to maximize interoperability with servers that are not able to deploy more secure alternatives.

Security Warning: In practice, many servers offer, and many clients use, TLS plus SASL PLAIN. The SCRAM-SHA-1 and especially SCRAM-SHA-1-PLUS variants of the SCRAM mechanism are strongly preferred over the PLAIN mechanism because of their superior security properties (including for SCRAM-SHA-1-PLUS the ability to enforce channel binding as described under Section 13.9.4). A client SHOULD treat TLS plus SASL PLAIN as a technology of last resort to be used only when interacting with a server that does not offer SCRAM (or other alternatives that are more secure than TLS plus SASL PLAIN), MUST prefer more secure mechanisms (e.g., EXTERNAL, SCRAM-SHA-1-PLUS, SCRAM-SHA-1, or the older DIGEST-MD5 mechanism) to the PLAIN mechanism, and MUST NOT use the PLAIN mechanism if the stream does not at a minimum have confidentiality and integrity protection via TLS with full certificate validation as described under Section 13.7.2.1. A server MUST NOT offer SASL PLAIN if the confidentiality and integrity of the stream are not protected via TLS or an equivalent security layer. A server SHOULD NOT offer TLS plus SASL PLAIN unless it is unable to offer some variant of SASL SCRAM (or other alternatives that are more secure than TLS plus SASL PLAIN), e.g., because the XMPP service depends for authentication purposes on a database or directory that is not under the control of the XMPP administrators, such as Pluggable Authentication Modules (PAM), an Lightweight Directory Access Protocol (LDAP) directory [LDAP], or an Authentication, Authorization, and Accounting (AAA) key management protocol (for guidance, refer to [AAA]). However, offering TLS plus SASL PLAIN even when the server supports more secure alternatives might be appropriate if the server needs to enable interoperability with an installed base of clients that do not yet support SCRAM or other alternatives that are more secure than TLS plus SASL PLAIN.

13.8.4. For Confidentiality and Authentication without Passwords

For both confidentiality and authentication without passwords, servers MUST and clients SHOULD implement TLS with the TLS_RSA_WITH_AES_128_CBC_SHA ciphersuite plus the SASL EXTERNAL mechanism (see Appendix A of [SASL]) with PKIX certificates.

13.9. Technology Reuse

13.9.1. Use of Base 64 in SASL

Both the client and the server MUST verify any base 64 data received during SASL negotiation (Section 6). An implementation MUST reject (not ignore) any characters that are not explicitly allowed by the base 64 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 base 64 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.

While base 64 encoding visually hides otherwise easily recognized information (such as passwords), it does not provide any computational confidentiality.

All uses of base 64 encoding MUST follow the definition in Section 4 of [BASE64] and padding bits MUST be set to zero.

13.9.2. Use of DNS

XMPP typically relies on the Domain Name System (specifically [DNS-SRV] records) to resolve a fully qualified domain name to an IP address before a client connects to a server or before a peer server connects to another server. Before attempting to negotiate an XML stream, the initiating entity MUST NOT proceed until it has resolved the DNS domain name of the receiving entity as specified under Section 3 (although it is not necessary to resolve the DNS domain name before each connection attempt, because DNS resolution results can be temporarily cached in accordance with time-to-live values). However, in the absence of a secure DNS option (e.g., as provided by [DNSSEC]), a malicious attacker with access to the DNS server data, or able to cause spoofed answers to be cached in a recursive resolver, can potentially cause the initiating entity to connect to any XMPP server chosen by the attacker. Deployment and validation of server certificates help to prevent such attacks.

13.9.3. Use of Hash Functions

XMPP itself does not directly mandate the use of any particular cryptographic hash function. However, technologies on which XMPP depends (e.g., TLS and particular SASL mechanisms), as well as various XMPP extensions, might make use of cryptographic hash functions. Those who implement XMPP technologies or who develop XMPP extensions are advised to closely monitor the state of the art regarding attacks against cryptographic hash functions in Internet protocols as they relate to XMPP. For helpful guidance, refer to [HASHES].

13.9.4. Use of SASL

Because the initiating entity chooses an acceptable SASL mechanism from the list presented by the receiving entity, the initiating entity depends on the receiving entity's list for authentication. This dependency introduces the possibility of a downgrade attack if an attacker can gain control of the channel and therefore present a weak list of mechanisms. To mitigate this attack, the parties SHOULD protect the channel using TLS before attempting SASL negotiation and either perform full certificate validation as described under Section 13.7.2.1 or use a SASL mechanism that provides channel bindings, such as SCRAM-SHA-1-PLUS. (Protecting the channel via TLS with full certificate validation can help to ensure the confidentiality and integrity of the information exchanged during SASL negotiation.)

The SASL framework itself does not provide a method for binding SASL authentication to a security layer providing confidentiality and integrity protection that was negotiated at a lower layer (e.g., TLS). Such a binding is known as a "channel binding" (see [CHANNEL]). Some SASL mechanisms provide channel bindings, which in the case of XMPP would typically be a binding to TLS (see [CHANNEL-TLS]). If a SASL mechanism provides a channel binding (e.g., this is true of [SCRAM]), then XMPP entities using that mechanism SHOULD prefer the channel binding variant (e.g., preferring "SCRAM-SHA-1-PLUS" over "SCRAM-SHA-1"). If a SASL mechanism does not provide a channel binding, then the mechanism cannot provide a way 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; furthermore, if the end points are not identical, then 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.

Many deployed XMPP services authenticate client connections by means of passwords. It is well known that most human users choose relatively weak passwords. Although service provisioning is out of scope for this document, XMPP servers that allow password-based authentication SHOULD enforce minimal criteria for password strength to help prevent dictionary attacks. Because all password-based authentication mechanisms are susceptible to password guessing attacks, XMPP servers MUST limit the number of retries allowed during SASL authentication, as described under Section 6.4.5.

Some SASL mechanisms (e.g., [ANONYMOUS]) do not provide strong peer entity authentication of the client to the server. Service administrators are advised to enable such mechanisms with caution. Best practices for the use of the SASL ANONYMOUS mechanism in XMPP are described in [XEP-0175].

13.9.5. Use of TLS

Implementations of TLS typically support multiple versions of the Transport Layer Security protocol as well as the older Secure Sockets Layer (SSL) protocol. Because of known security vulnerabilities, XMPP servers and clients MUST NOT request, offer, or use SSL 2.0. For further details, see Appendix E.2 of [TLS] along with [TLS-SSL2].

To prevent man-in-the-middle attacks, the TLS client (which might be an XMPP client or an XMPP server) MUST verify the certificate of the TLS server and MUST check its understanding of the server FQDN against the server's identity as presented in the TLS Certificate message as described under Section 13.7.2.1 (for further details, see [TLS-CERTS].

Support for TLS renegotiation is strictly OPTIONAL. However, implementations that support TLS renegotiation MUST implement and use the TLS Renegotiation Extension [TLS-NEG]. Further details are provided under Section 5.3.5.

13.9.6. Use of UTF-8

The use of UTF-8 makes it possible to transport non-ASCII characters, and thus enables character "spoofing" scenarios, in which a displayed value appears to be something other than it is. Furthermore, there are known attack scenarios related to the decoding of UTF-8 data. On both of these points, refer to [UTF-8] for more information.

13.9.7. Use of XML

Because XMPP is an application profile of the Extensible Markup Language [XML], many of the security considerations described in [XML-MEDIA] and [XML-GUIDE] also apply to XMPP. Several aspects of XMPP mitigate the risks described there, such as the prohibitions specified under Section 11.1 and the lack of external references to style sheets or transformations, but these mitigating factors are by no means comprehensive.

13.10. Information Leaks

13.10.1. IP Addresses

A client's IP address and method of access MUST NOT be made public by a server (e.g., as typically occurs in [IRC]).

13.10.2. Presence Information

One of the core aspects of XMPP is presence: information about the network availability of an XMPP entity (i.e., whether the entity is currently online or offline). A "presence leak" occurs when an entity's network availability is inadvertently and involuntarily revealed to a second entity that is not authorized to know the first entity's network availability.

Although presence is discussed more fully in [XMPP-IM], it is important to note that an XMPP server MUST NOT leak presence. In particular at the core XMPP level, real-time addressing and network availability is associated with a specific connected resource; therefore, any disclosure of a connected resource's full JID comprises a presence leak. To help prevent such a presence leak, a server MUST NOT return different stanza errors depending on whether a potential attacker sends XML stanzas to the entity's bare JID (<localpart@domainpart>) or full JID (<localpart@domainpart/resourcepart>).

13.11. Directory Harvesting

If a server generates an error stanza in response to receiving a stanza for a user account that does not exist, using the <service-unavailable/> stanza error condition (Section 8.3.3.19) can help protect against directory harvesting attacks, since this is the same error condition that is returned if, for instance, the namespace of an IQ child element is not understood, or if "offline message storage" ([XEP-0160]) or message forwarding is not enabled for a domain. However, subtle differences in the exact XML of error stanzas, as well as in the timing with which such errors are returned, can enable an attacker to determine the network presence of a user when more advanced blocking technologies are not used (see for instance [XEP-0016] and [XEP-0191]). Therefore, a server that exercises a higher level of caution might not return any error at all in response to certain kinds of received stanzas, so that a non-existent user appears to behave like a user that has no interest in conversing with the sender.

13.12. Denial of Service

[DOS] defines denial of service as follows:

> A denial-of-service (DoS) attack is an attack in which one or more machines target a victim and attempt to prevent the victim from doing useful work. The victim can be a network server, client or router, a network link or an entire network, an individual Internet user or a company doing business using the Internet, an Internet Service Provider (ISP), country, or any combination of or variant on these.

Some considerations discussed in this document help to prevent denial-of-service attacks (e.g., the mandate that a server MUST NOT process XML stanzas from clients that have not yet provided appropriate authentication credentials and MUST NOT process XML stanzas from peer servers whose identity it has not either authenticated via SASL or weakly verified via Server Dialback).

In addition, [XEP-0205] provides a detailed discussion of potential denial-of-service attacks against XMPP systems along with best practices for preventing such attacks. The recommendations include:

1. A server implementation SHOULD enable a server administrator to limit the number of TCP connections that it will accept from a given IP address at any one time. If an entity attempts to connect but the maximum number of TCP connections has been reached, the receiving server MUST NOT allow the new connection to proceed.

2. A server implementation SHOULD enable a server administrator to limit the number of TCP connection attempts that it will accept from a given IP address in a given time period. If an entity attempts to connect but the maximum number of connection attempts has been reached, the receiving server MUST NOT allow the new connection to proceed.

3. A server implementation SHOULD enable a server administrator to limit the number of connected resources it will allow an account to bind at any one time. If a client attempts to bind a resource

but it has already reached the configured number of allowable resources, the receiving server MUST return a <resource-constraint/> stanza error (Section 8.3.3.18).

4. A server implementation SHOULD enable a server administrator to limit the size of stanzas it will accept from a connected client or peer server (where "size" is inclusive of all XML markup as defined in Section 2.4 of [XML], from the opening "<" character of the stanza to the closing ">" character). A deployed server's maximum stanza size MUST NOT be smaller than 10000 bytes, which reflects a reasonable compromise between the benefits of expressiveness for originating entities and the costs of stanza processing for servers. A server implementation SHOULD NOT blindly set 10000 bytes as the value for all deployments but instead SHOULD enable server administrators to set their own limits. If a connected resource or peer server sends a stanza that violates the upper limit, the receiving server MUST either return a <policy-violation/> stanza error (Section 8.3.3.12), thus allowing the sender to recover, or close the stream with a <policy-violation/> stream error (Section 4.9.3.14).

5. A server implementation SHOULD enable a server administrator to limit the number of XML stanzas that a connected client is allowed to send to distinct recipients within a given time period. If a connected client sends too many stanzas to distinct recipients in a given time period, the receiving server SHOULD NOT process the stanza and instead SHOULD return a <policy-violation/> stanza error (Section 8.3.3.12).

6. A server implementation SHOULD enable a server administrator to limit the amount of bandwidth it will allow a connected client or peer server to use in a given time period.

7. A server implementation MAY enable a server administrator to limit the types of stanzas (based on the extended content "payload") that it will allow a connected resource or peer server send over an active connection. Such limits and restrictions are a matter of deployment policy.

8. A server implementation MAY refuse to route or deliver any stanza that it considers to be abusive, with or without returning an error to the sender.

For more detailed recommendations regarding denial-of-service attacks in XMPP systems, refer to [XEP-0205].

13.13. Firewalls

Although DNS SRV records can instruct connecting entities to use TCP ports other than 5222 (client-to-server) and 5269 (server-to-server), communication using XMPP typically occurs over those ports, which are registered with the IANA (see Section 14). Use of these well-known ports allows administrators to easily enable or disable XMPP activity through existing and commonly deployed firewalls.

13.14. Interdomain Federation

The term "federation" is commonly used to describe communication between two servers.

Because service provisioning is a matter of policy, it is OPTIONAL for any given server to support federation. If a particular server enables federation, it SHOULD enable strong security as previously described to ensure both authentication and confidentiality; compliant implementations SHOULD support TLS and SASL for this purpose.

Before RFC 3920 defined TLS plus SASL EXTERNAL with certificates for encryption and authentication of server-to-server streams, the only method for weak identity verification of a peer server was Server Dialback as defined in [XEP-0220]. Even when [DNSSEC] is used, Server Dialback provides only weak identity verification and provides no confidentiality or integrity. At the time of writing, Server Dialback is still the most widely used technique for some level of assurance over server-to-server streams. This reality introduces the possibility of a downgrade attack from TLS + SASL EXTERNAL to Server Dialback if an attacker can gain control of the channel and therefore convince the initiating server that the receiving server does not support TLS or does not have an appropriate certificate. To help prevent this attack, the parties SHOULD protect the channel using TLS before proceeding, even if the presented certificates are self-signed or otherwise untrusted.

13.15. Non-Repudiation

Systems that provide both peer entity authentication and data integrity have the potential to enable an entity to prove to a third party that another entity intended to send particular data. Although XMPP systems can provide both peer entity authentication and data integrity, XMPP was never designed to provide non-repudiation.

14. IANA Considerations

The following subsections update the registrations provided in [RFC3920]. This section is to be interpreted according to [IANA-GUIDE].

14.1. XML Namespace Name for TLS Data

A URN sub-namespace for STARTTLS negotiation 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-tls
Specification: RFC 6120
Description: This is the XML namespace name for STARTTLS negotiation data in the Extensible Messaging and Presence Protocol (XMPP) as defined by RFC 6120.
Registrant Contact: IESG <iesg@ietf.org>

14.2. XML Namespace Name for SASL Data

A URN sub-namespace for SASL negotiation 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 6120
Description: This is the XML namespace name for SASL negotiation data in the Extensible Messaging and Presence Protocol (XMPP) as defined by RFC 6120.
Registrant Contact: IESG <iesg@ietf.org>

14.3. XML Namespace Name for Stream Errors

A URN sub-namespace for stream 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 6120
Description: This is the XML namespace name for stream error data in the Extensible Messaging and Presence Protocol (XMPP) as defined by RFC 6120.
Registrant Contact: IESG <iesg@ietf.org>

14.4. 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 6120
Description: This is the XML namespace name for resource binding in the Extensible Messaging and Presence Protocol (XMPP) as defined by RFC 6120.
Registrant Contact: IESG <iesg@ietf.org>

14.5. XML Namespace Name for Stanza Errors

A URN sub-namespace for stanza 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 6120
Description: This is the XML namespace name for stanza error data in the Extensible Messaging and Presence Protocol (XMPP) as defined by RFC 6120.
Registrant Contact: IESG <iesg@ietf.org>

14.6. GSSAPI Service Name

The IANA has registered "xmpp" as a [GSS-API] service name, as defined under Section 6.6.

14.7. Port Numbers and Service Names

The IANA has registered "xmpp-client" and "xmpp-server" as keywords for [TCP] ports 5222 and 5269, respectively. In accordance with [IANA-PORTS], this document updates the existing registration, as follows.

Service Name: xmpp-client
Transport Protocol: TCP
Description: A service offering support for connections by XMPP client applications
Registrant: IETF XMPP Working Group
Contact: IESG <iesg@ietf.org>
Reference: RFC 6120
Port Number: 5222

Service Name: xmpp-server
Transport Protocol: TCP
Description: A service offering support for connections by XMPP server applications
Registrant: IETF XMPP Working Group
Contact: IESG <iesg@ietf.org>
Reference: RFC 6120
Port Number: 5269

15. Conformance Requirements

This section describes a protocol feature set that summarizes the conformance requirements of this specification. This feature set is appropriate for use in software certification, interoperability testing, and implementation reports. For each feature, this section provides the following information:

- A human-readable name
- An informational description
- A reference to the particular section of this document that normatively defines the feature
- Whether the feature applies to the Client role, the Server role, or both (where "N/A" signifies that the feature is not applicable to the specified role)
- Whether the feature MUST or SHOULD be implemented, where the capitalized terms are to be understood as described in [KEYWORDS]

The feature set specified here attempts to adhere to the concepts and formats proposed by Larry Masinter within the IETF's NEWTRK Working Group in 2005, as captured in [INTEROP]. Although this feature set is more detailed than called for by [REPORTS], it provides a suitable basis for the generation of implementation reports to be submitted in support of advancing this specification from Proposed Standard to Draft Standard in accordance with [PROCESS].

Feature: bind-gen
Description: Generate a random resource on demand.
Section: Section 7.6
Roles: Client N/A, Server MUST.

Feature: bind-mtn
Description: Consider resource binding as mandatory-to-negotiate.
Section: Section 7.3.1
Roles: Client MUST, Server MUST.

```
   Feature:  bind-restart
   Description:  Do not restart the stream after negotiation of resource
      binding.
   Section:  Section 7.3.2
   Roles:  Client MUST, Server MUST.

   Feature:  bind-support
   Description:  Support binding of client resources to an authenticated
      stream.
   Section:  Section 7
   Roles:  Client MUST, Server MUST.

   Feature:  sasl-correlate
   Description:  When authenticating a stream peer using SASL, correlate
      the authentication identifier resulting from SASL negotiation with
      the 'from' address (if any) of the stream header it received from
      the peer.
   Section:  Section 6.4.6
   Roles:  Client SHOULD, Server SHOULD.

   Feature:  sasl-errors
   Description:  Support SASL errors during the negotiation process.
   Section:  Section 6.5
   Roles:  Client MUST, Server MUST.

   Feature:  sasl-mtn
   Description:  Consider SASL as mandatory-to-negotiate.
   Section:  Section 6.3.1
   Roles:  Client MUST, Server MUST.

   Feature:  sasl-restart
   Description:  Initiate or handle a stream restart after SASL
      negotiation.
   Section:  Section 6.3.2
   Roles:  Client MUST, Server MUST.

   Feature:  sasl-support
   Description:  Support the Simple Authentication and Security Layer
      for stream authentication.
   Section:  Section 6
   Roles:  Client MUST, Server MUST.

   Feature:  security-mti-auth-scram
   Description:  Support the SASL SCRAM mechanism for authentication
      only (this implies support for both the SCRAM-SHA-1 and
      SCRAM-SHA-1-PLUS variants).
   Section:  Section 13.8
   Roles:  Client MUST, Server MUST.
   Feature:  security-mti-both-external
   Description:  Support TLS with SASL EXTERNAL for confidentiality and
      authentication.
   Section:  Section 13.8
   Roles:  Client SHOULD, Server MUST.

   Feature:  security-mti-both-plain
   Description:  Support TLS using the TLS_RSA_WITH_AES_128_CBC_SHA
      ciphersuite plus the SASL PLAIN mechanism for confidentiality and
      authentication.
   Section:  Section 13.8
   Roles:  Client SHOULD, Server MAY.

   Feature:  security-mti-both-scram
   Description:  Support TLS using the TLS_RSA_WITH_AES_128_CBC_SHA
      ciphersuite plus the SCRAM-SHA-1 and SCRAM-SHA-1-PLUS variants of
      the SASL SCRAM mechanism for confidentiality and authentication.
   Section:  Section 13.8
   Roles:  Client MUST, Server MUST.

   Feature:  security-mti-confidentiality
   Description:  Support TLS using the TLS_RSA_WITH_AES_128_CBC_SHA
      ciphersuite for confidentiality only.
   Section:  Section 13.8
   Roles:  Client N/A, Server SHOULD.

   Feature:  stanza-attribute-from
   Description:  Support the common 'from' attribute for all stanza
      kinds.
   Section:  Section 8.1.2
   Roles:  Client MUST, Server MUST.

   Feature:  stanza-attribute-from-stamp
   Description:  Stamp or rewrite the 'from' address of all stanzas
      received from connected clients.
   Section:  Section 8.1.2.1
   Roles:  Client N/A, Server MUST.

   Feature:  stanza-attribute-from-validate
   Description:  Validate the 'from' address of all stanzas received
      from peer servers.
   Section:  Section 8.1.2.2
   Roles:  Client N/A, Server MUST.

   Feature:  stanza-attribute-id
   Description:  Support the common 'id' attribute for all stanza kinds.
   Section:  Section 8.1.3
   Roles:  Client MUST, Server MUST.
   Feature:  stanza-attribute-to
   Description:  Support the common 'to' attribute for all stanza kinds.
   Section:  Section 8.1.1
   Roles:  Client MUST, Server MUST.

   Feature:  stanza-attribute-to-validate
   Description:  Ensure that all stanzas received from peer servers
      include a 'to' address.
   Section:  Section 8.1.1
   Roles:  Client N/A, Server MUST.

   Feature:  stanza-attribute-type
   Description:  Support the common 'type' attribute for all stanza
      kinds.
   Section:  Section 8.1.4
   Roles:  Client MUST, Server MUST.

   Feature:  stanza-attribute-xmllang
   Description:  Support the common 'xml:lang' attribute for all stanza
      kinds.
   Section:  Section 8.1.5
   Roles:  Client MUST, Server MUST.

   Feature:  stanza-error
   Description:  Generate and handle stanzas of type "error" for all
      stanza kinds.
   Section:  Section 8.3
   Roles:  Client MUST, Server MUST.

   Feature:  stanza-error-child
   Description:  Ensure that stanzas of type "error" include an <error/>
      child element.
   Section:  Section 8.3
   Roles:  Client MUST, Server MUST.

   Feature:  stanza-error-id
   Description:  Ensure that stanzas of type "error" preserve the 'id'
      provided in the triggering stanza.
   Section:  Section 8.3
   Roles:  Client MUST, Server MUST.

   Feature:  stanza-error-reply
   Description:  Do not reply to a stanza of type "error" with another
      stanza of type "error".
   Section:  Section 8.3
   Roles:  Client MUST, Server MUST.
   Feature:  stanza-extension
   Description:  Correctly process XML data qualified by an unsupported
      XML namespace, where "correctly process" means to ignore that
      portion of the stanza in the case of a message or presence stanza
      and return an error in the case of an IQ stanza (for the intended
      recipient), and to route or deliver the stanza (for a routing
      entity such as a server).
   Section:  Section 8.4
   Roles:  Client MUST, Server MUST.

   Feature:  stanza-iq-child
   Description:  Include exactly one child element in an <iq/> stanza of
      type "get" or "set", zero or one child elements in an <iq/> stanza
      of type "result", and one or two child elements in an <iq/> stanza
      of type "error".
   Section:  Section 8.2.3
   Roles:  Client MUST, Server MUST.

   Feature:  stanza-iq-id
   Description:  Ensure that all <iq/> stanzas include an 'id'
      attribute.
   Section:  Section 8.2.3
   Roles:  Client MUST, Server MUST.

   Feature:  stanza-iq-reply
   Description:  Reply to an <iq/> stanza of type "get" or "set" with an
      <iq/> stanza of type "result" or "error".
   Section:  Section 8.2.3
   Roles:  Client MUST, Server MUST.

   Feature:  stanza-iq-type
   Description:  Ensure that all <iq/> stanzas include a 'type'
      attribute whose value is "get", "set", "result", or "error".
   Section:  Section 8.2.3
   Roles:  Client MUST, Server MUST.

   Feature:  stanza-kind-iq
   Description:  Support the <iq/> stanza.
   Section:  Section 8.2.3
   Roles:  Client MUST, Server MUST.

   Feature:  stanza-kind-message
   Description:  Support the <message/> stanza.
   Section:  Section 8.2.1
   Roles:  Client MUST, Server MUST.
   Feature:  stanza-kind-presence
   Description:  Support the <presence/> stanza.
   Section:  Section 8.2.2
   Roles:  Client MUST, Server MUST.

   Feature:  stream-attribute-initial-from
   Description:  Include a 'from' attribute in the initial stream
      header.
   Section:  Section 4.7.1
   Roles:  Client SHOULD, Server MUST.

   Feature:  stream-attribute-initial-lang
   Description:  Include an 'xml:lang' attribute in the initial stream
      header.
   Section:  Section 4.7.4
   Roles:  Client SHOULD, Server SHOULD.

   Feature:  stream-attribute-initial-to
   Description:  Include a 'to' attribute in the initial stream header.
   Section:  Section 4.7.2
   Roles:  Client MUST, Server MUST.

   Feature:  stream-attribute-response-from
   Description:  Include a 'from' attribute in the response stream
      header.
   Section:  Section 4.7.1
   Roles:  Client N/A, Server MUST.

   Feature:  stream-attribute-response-id
   Description:  Include an 'id' attribute in the response stream
      header.
   Section:  Section 4.7.3
   Roles:  Client N/A, Server MUST.

   Feature:  stream-attribute-response-id-unique
   Description:  Ensure that the 'id' attribute in the response stream
      header is unique within the context of the receiving entity.
   Section:  Section 4.7.3
   Roles:  Client N/A, Server MUST.

   Feature:  stream-attribute-response-to
   Description:  Include a 'to' attribute in the response stream header.
   Section:  Section 4.7.2
   Roles:  Client N/A, Server SHOULD.
```

Feature: stream-error-generate
Description: Generate a stream error (followed by a closing stream tag and termination of the TCP connection) upon detecting a stream-related error condition.
Section: Section 4.9
Roles: Client MUST, Server MUST.

Feature: stream-fqdn-resolution
Description: Resolve FQDNs before opening a TCP connection to the receiving entity.
Section: Section 3.2
Roles: Client MUST, Server MUST.

Feature: stream-negotiation-complete
Description: Do not consider the stream negotiation process to be complete until the receiving entity sends a stream features advertisement that is empty or that contains only voluntary-to-negotiate features.
Section: Section 4.3.5
Roles: Client MUST, Server MUST.

Feature: stream-negotiation-features
Description: Send stream features after sending a response stream header.
Section: Section 4.3.2
Roles: Client N/A, Server MUST.

Feature: stream-negotiation-restart
Description: Consider the previous stream to be replaced upon negotiation of a stream feature that necessitates a stream restart, and send or receive a new initial stream header after negotiation of such a stream feature.
Section: Section 4.3.3
Roles: Client MUST, Server MUST.

Feature: stream-reconnect
Description: Reconnect with exponential backoff if a TCP connection is terminated unexpectedly.
Section: Section 3.3
Roles: Client MUST, Server MUST.

Feature: stream-tcp-binding
Description: Bind an XML stream to a TCP connection.
Section: Section 3
Roles: Client MUST, Server MUST.

```
   Feature:  tls-certs
   Description:  Check the identity specified in a certificate that is
      presented during TLS negotiation.
   Section:  Section 13.7.2
   Roles:  Client MUST, Server MUST.

   Feature:  tls-mtn
   Description:  Consider TLS as mandatory-to-negotiate if STARTTLS is
      the only feature advertised or if the STARTTLS feature
      advertisement includes an empty <required/> element.
   Section:  Section 5.3.1
   Roles:  Client MUST, Server MUST.

   Feature:  tls-restart
   Description:  Initiate or handle a stream restart after TLS
      negotiation.
   Section:  Section 5.3.2
   Roles:  Client MUST, Server MUST.

   Feature:  tls-support
   Description:  Support Transport Layer Security for stream encryption.
   Section:  Section 5
   Roles:  Client MUST, Server MUST.

   Feature:  tls-correlate
   Description:  When validating a certificate presented by a stream
      peer during TLS negotiation, correlate the validated identity with
      the 'from' address (if any) of the stream header it received from
      the peer.
   Section:  Section 13.7.2
   Roles:  Client SHOULD, Server SHOULD.

   Feature:  xml-namespace-content-client
   Description:  Support 'jabber:client' as a content namespace.
   Section:  Section 4.8.2
   Roles:  Client MUST, Server MUST.

   Feature:  xml-namespace-content-server
   Description:  Support 'jabber:server' as a content namespace.
   Section:  Section 4.8.2
   Roles:  Client N/A, Server MUST.

   Feature:  xml-namespace-streams-declaration
   Description:  Ensure that there is a namespace declaration for the
      'http://etherx.jabber.org/streams' namespace.
   Section:  Section 4.8.1
   Roles:  Client MUST, Server MUST.
```

Feature: xml-namespace-streams-prefix
Description: Ensure that all elements qualified by the 'http://etherx.jabber.org/streams' namespace are prefixed by the prefix (if any) defined in the namespace declaration.
Section: Section 4.8.1
Roles: Client MUST, Server MUST.

Feature: xml-restriction-comment
Description: Do not generate or accept XML comments.
Section: Section 11.1
Roles: Client MUST, Server MUST.

Feature: xml-restriction-dtd
Description: Do not generate or accept internal or external DTD subsets.
Section: Section 11.1
Roles: Client MUST, Server MUST.

Feature: xml-restriction-pi
Description: Do not generate or accept XML processing instructions.
Section: Section 11.1
Roles: Client MUST, Server MUST.

Feature: xml-restriction-ref
Description: Do not generate or accept internal or external entity references with the exception of the predefined entities.
Section: Section 11.1
Roles: Client MUST, Server MUST.

Feature: xml-wellformed-xml
Description: Do not generate or accept data that is not XML-well-formed.
Section: Section 11.3
Roles: Client MUST, Server MUST.

Feature: xml-wellformed-ns
Description: Do not generate or accept data that is not namespace-well-formed.
Section: Section 11.3
Roles: Client MUST, Server MUST.

16. References

16.1. Normative References

[BASE64]   Josefsson, S., "The Base16, Base32, and Base64 Data Encodings", RFC 4648, October 2006.

[CHANNEL]   Williams, N., "On the Use of Channel Bindings to Secure Channels", RFC 5056, November 2007.

[CHANNEL-TLS]   Altman, J., Williams, N., and L. Zhu, "Channel Bindings for TLS", RFC 5929, July 2010.

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

[DNS-CONCEPTS]   Mockapetris, P., "Domain names - concepts and facilities", STD 13, RFC 1034, November 1987.

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

[IPv6-ADDR]   Kawamura, S. and M. Kawashima, "A Recommendation for IPv6 Address Text Representation", RFC 5952, August 2010.

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

[LANGMATCH]   Phillips, A. and M. Davis, "Matching of Language Tags", BCP 47, RFC 4647, September 2006.

[LANGTAGS]   Phillips, A. and M. Davis, "Tags for Identifying Languages", BCP 47, RFC 5646, September 2009.

[OCSP]   Myers, M., Ankney, R., Malpani, A., Galperin, S., and C. Adams, "X.509 Internet Public Key Infrastructure Online Certificate Status Protocol - OCSP", RFC 2560, June 1999.

[PKIX]   Cooper, D., Santesson, S., Farrell, S., Boeyen, S., Housley, R., and W. Polk, "Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile", RFC 5280, May 2008.

[PKIX-ALGO]	Jonsson, J. and B. Kaliski, "Public-Key Cryptography Standards (PKCS) #1: RSA Cryptography Specifications Version 2.1", RFC 3447, February 2003.
[PKIX-SRV]	Santesson, S., "Internet X.509 Public Key Infrastructure Subject Alternative Name for Expression of Service Name", RFC 4985, August 2007.
[PLAIN]	Zeilenga, K., "The PLAIN Simple Authentication and Security Layer (SASL) Mechanism", RFC 4616, August 2006.
[RANDOM]	Eastlake, D., Schiller, J., and S. Crocker, "Randomness Requirements for Security", BCP 106, RFC 4086, June 2005.
[SASL]	Melnikov, A. and K. Zeilenga, "Simple Authentication and Security Layer (SASL)", RFC 4422, June 2006.
[SCRAM]	Newman, C., Menon-Sen, A., Melnikov, A., and N. Williams, "Salted Challenge Response Authentication Mechanism (SCRAM) SASL and GSS-API Mechanisms", RFC 5802, July 2010.
[STRONGSEC]	Schiller, J., "Strong Security Requirements for Internet Engineering Task Force Standard Protocols", BCP 61, RFC 3365, August 2002.
[TCP]	Postel, J., "Transmission Control Protocol", STD 7, RFC 793, September 1981.
[TLS]	Dierks, T. and E. Rescorla, "The Transport Layer Security (TLS) Protocol Version 1.2", RFC 5246, August 2008.
[TLS-CERTS]	Saint-Andre, P. and J. Hodges, "Representation and Verification of Domain-Based Application Service Identity within Internet Public Key Infrastructure Using X.509 (PKIX) Certificates in the Context of Transport Layer Security (TLS)", RFC 6125, March 2011.
[TLS-NEG]	Rescorla, E., Ray, M., Dispensa, S., and N. Oskov, "Transport Layer Security (TLS) Renegotiation Indication Extension", RFC 5746, February 2010.
[TLS-SSL2]	Turner, S. and T. Polk, "Prohibiting Secure Sockets Layer (SSL) Version 2.0", RFC 6176, March 2011.

[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.

[UNICODE] The Unicode Consortium, "The Unicode Standard, Version 6.0", 2010, <http://www.unicode.org/versions/Unicode6.0.0/>.

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

[URI] Berners-Lee, T., Fielding, R., and L. Masinter, "Uniform Resource Identifier (URI): Generic Syntax", STD 66, RFC 3986, January 2005.

[X509] International Telecommunications Union, "Information technology - Open Systems Interconnection - The Directory: Public-key and attribute certificate frameworks", ITU-T Recommendation X.509, ISO Standard 9594-8, March 2000.

[XML] Maler, E., Yergeau, F., Sperberg-McQueen, C., Paoli, J., and T. Bray, "Extensible Markup Language (XML) 1.0 (Fifth Edition)", World Wide Web Consortium Recommendation REC-xml-20081126, November 2008, <http://www.w3.org/TR/2008/REC-xml-20081126>.

[XML-GUIDE] Hollenbeck, S., Rose, M., and L. Masinter, "Guidelines for the Use of Extensible Markup Language (XML) within IETF Protocols", BCP 70, RFC 3470, January 2003.

[XML-MEDIA] Murata, M., St. Laurent, S., and D. Kohn, "XML Media Types", RFC 3023, January 2001.

[XML-NAMES] Thompson, H., Hollander, D., Layman, A., Bray, T., and R. Tobin, "Namespaces in XML 1.0 (Third Edition)", World Wide Web Consortium Recommendation REC-xml-names-20091208, December 2009, <http://www.w3.org/TR/2009/REC-xml-names-20091208>.

[XMPP-ADDR] Saint-Andre, P., "Extensible Messaging and Presence Protocol (XMPP): Address Format", RFC 6122, March 2011.

[XMPP-IM] Saint-Andre, P., "Extensible Messaging and Presence Protocol (XMPP): Instant Messaging and Presence", RFC 6121, March 2011.

16.2. Informative References

[AAA] Housley, R. and B. Aboba, "Guidance for Authentication, Authorization, and Accounting (AAA) Key Management", BCP 132, RFC 4962, July 2007.

[ABNF] Crocker, D. and P. Overell, "Augmented BNF for Syntax Specifications: ABNF", STD 68, RFC 5234, January 2008.

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

[ANONYMOUS] Zeilenga, K., "Anonymous Simple Authentication and Security Layer (SASL) Mechanism", RFC 4505, June 2006.

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

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

[DNSSEC] Arends, R., Austein, R., Larson, M., Massey, D., and S. Rose, "DNS Security Introduction and Requirements", RFC 4033, March 2005.

[DNS-TXT] Rosenbaum, R., "Using the Domain Name System To Store Arbitrary String Attributes", RFC 1464, May 1993.

[DOS] Handley, M., Rescorla, E., and IAB, "Internet Denial-of-Service Considerations", RFC 4732, December 2006.

[E2E-REQS] Saint-Andre, P., "Requirements for End-to-End Encryption in the Extensible Messaging and Presence Protocol (XMPP)", Work in Progress, March 2010.

[EMAIL-ARCH] Crocker, D., "Internet Mail Architecture", RFC 5598, July 2009.

[ETHERNET] "Information technology - Telecommunications and information exchange between systems - Local and metropolitan area networks - Specific requirements - Part 3: Carrier sense multiple access with collision detection (CSMA/CD) access method and physical layer specifications", IEEE Standard 802.3, September 1998.

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

[HASHES] Hoffman, P. and B. Schneier, "Attacks on Cryptographic Hashes in Internet Protocols", RFC 4270, November 2005.

[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.

[IANA-GUIDE] Narten, T. and H. Alvestrand, "Guidelines for Writing an IANA Considerations Section in RFCs", BCP 26, RFC 5226, May 2008.

[IANA-PORTS] Cotton, M., Eggert, L., Touch, J., Westerlund, M., and S. Cheshire, "Internet Assigned Numbers Authority (IANA) Procedures for the Management of the Transport Protocol Port Number and Service Name Registry", Work in Progress, February 2011.

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

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

[INTEROP] Masinter, L., "Formalizing IETF Interoperability Reporting", Work in Progress, October 2005.

[IRC] Kalt, C., "Internet Relay Chat: Architecture", RFC 2810, April 2000.

[IRI] Duerst, M. and M. Suignard, "Internationalized Resource Identifiers (IRIs)", RFC 3987, January 2005.

[LDAP] Zeilenga, K., "Lightweight Directory Access Protocol (LDAP): Technical Specification Road Map", RFC 4510, June 2006.

[LINKLOCAL] Cheshire, S., Aboba, B., and E. Guttman, "Dynamic Configuration of IPv4 Link-Local Addresses", RFC 3927, May 2005.

[MAILBOXES] Crocker, D., "MAILBOX NAMES FOR COMMON SERVICES, ROLES AND FUNCTIONS", RFC 2142, May 1997.

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

[PROCESS] Bradner, S., "The Internet Standards Process -- Revision 3", BCP 9, RFC 2026, October 1996.

[REPORTS] Dusseault, L. and R. Sparks, "Guidance on Interoperation and Implementation Reports for Advancement to Draft Standard", BCP 9, RFC 5657, September 2009.

[REST] Fielding, R., "Architectural Styles and the Design of Network-based Software Architectures", 2000.

[RFC3920] Saint-Andre, P., Ed., "Extensible Messaging and Presence Protocol (XMPP): Core", RFC 3920, October 2004.

[RFC3921] Saint-Andre, P., Ed., "Extensible Messaging and Presence Protocol (XMPP): Instant Messaging and Presence", RFC 3921, October 2004.

[SASLPREP] Zeilenga, K., "SASLprep: Stringprep Profile for User Names and Passwords", RFC 4013, February 2005.

[SEC-TERMS] Shirey, R., "Internet Security Glossary, Version 2", RFC 4949, August 2007.

[SMTP] Klensin, J., "Simple Mail Transfer Protocol", RFC 5321, October 2008.

[SEC-GUIDE] Rescorla, E. and B. Korver, "Guidelines for Writing RFC Text on Security Considerations", BCP 72, RFC 3552, July 2003.

[TLS-EXT] Eastlake 3rd, D., "Transport Layer Security (TLS) Extensions: Extension Definitions", RFC 6066, January 2011.

[TLS-RESUME] Salowey, J., Zhou, H., Eronen, P., and H. Tschofenig, "Transport Layer Security (TLS) Session Resumption without Server-Side State", RFC 5077, January 2008.

[URN-OID] Mealling, M., "A URN Namespace of Object Identifiers", RFC 3061, February 2001.

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

[UUID] Leach, P., Mealling, M., and R. Salz, "A Universally Unique IDentifier (UUID) URN Namespace", RFC 4122, July 2005.

[XEP-0001] Saint-Andre, P., "XMPP Extension Protocols", XSF XEP 0001, March 2010.

[XEP-0016] Millard, P. and P. Saint-Andre, "Privacy Lists", XSF XEP 0016, February 2007.

[XEP-0045] Saint-Andre, P., "Multi-User Chat", XSF XEP 0045, July 2007.

[XEP-0060] Millard, P., Saint-Andre, P., and R. Meijer, "Publish-Subscribe", XSF XEP 0060, July 2010.

[XEP-0071] Saint-Andre, P., "XHTML-IM", XSF XEP 0071, September 2008.

[XEP-0077] Saint-Andre, P., "In-Band Registration", XSF XEP 0077, September 2009.

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

[XEP-0100] Saint-Andre, P. and D. Smith, "Gateway Interaction", XSF XEP 0100, October 2005.

[XEP-0114] Saint-Andre, P., "Jabber Component Protocol", XSF XEP 0114, March 2005.

[XEP-0124] Paterson, I., Smith, D., and P. Saint-Andre, "Bidirectional-streams Over Synchronous HTTP (BOSH)", XSF XEP 0124, July 2010.

[XEP-0138] Hildebrand, J. and P. Saint-Andre, "Stream Compression", XSF XEP 0138, May 2009.

[XEP-0156] Hildebrand, J. and P. Saint-Andre, "Discovering Alternative XMPP Connection Methods", XSF XEP 0156, June 2007.

[XEP-0160] Saint-Andre, P., "Best Practices for Handling Offline Messages", XSF XEP 0160, January 2006.

[XEP-0174] Saint-Andre, P., "Link-Local Messaging", XSF XEP 0174, November 2008.

[XEP-0175] Saint-Andre, P., "Best Practices for Use of SASL ANONYMOUS", XSF XEP 0175, September 2009.

[XEP-0178] Saint-Andre, P. and P. Millard, "Best Practices for Use of SASL EXTERNAL with Certificates", XSF XEP 0178, February 2007.

[XEP-0191] Saint-Andre, P., "Simple Communications Blocking", XSF XEP 0191, February 2007.

[XEP-0198] Karneges, J., Hildebrand, J., Saint-Andre, P., Forno, F., Cridland, D., and M. Wild, "Stream Management", XSF XEP 0198, February 2011.

[XEP-0199] Saint-Andre, P., "XMPP Ping", XSF XEP 0199, June 2009.

[XEP-0205] Saint-Andre, P., "Best Practices to Discourage Denial of Service Attacks", XSF XEP 0205, January 2009.

[XEP-0206] Paterson, I. and P. Saint-Andre, "XMPP Over BOSH", XSF XEP 0206, July 2010.

[XEP-0220] Miller, J., Saint-Andre, P., and P. Hancke, "Server Dialback", XSF XEP 0220, March 2010.

[XEP-0225] Saint-Andre, P., "Component Connections", XSF XEP 0225, October 2008.

[XEP-0233] Miller, M., Saint-Andre, P., and J. Hildebrand, "Domain-Based Service Names in XMPP SASL Negotiation", XSF XEP 0233, June 2010.

[XEP-0288] Hancke, P. and D. Cridland, "Bidirectional Server-to-Server Connections", XSF XEP 0288, October 2010.

[XML-FRAG]   Grosso, P. and D. Veillard, "XML Fragment Interchange", World Wide Web Consortium CR CR-xml-fragment-20010212, February 2001, <http://www.w3.org/TR/2001/CR-xml-fragment-20010212>.

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

[XML-SCHEMA]   Thompson, H., Maloney, M., Mendelsohn, N., and D. Beech, "XML Schema Part 1: Structures Second Edition", World Wide Web Consortium Recommendation REC-xmlschema-1-20041028, October 2004, <http://www.w3.org/TR/2004/REC-xmlschema-1-20041028>.

[XMPP-URI]   Saint-Andre, P., "Internationalized Resource Identifiers (IRIs) and Uniform Resource Identifiers (URIs) for the Extensible Messaging and Presence Protocol (XMPP)", RFC 5122, February 2008.

Appendix A. XML Schemas

The following schemas formally define various namespaces used in this document, in conformance with [XML-SCHEMA]. Because validation of XML streams and stanzas is optional, these schemas are not normative and are provided for descriptive purposes only.

A.1. Stream 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:import namespace='jabber:client'/>
     <xs:import namespace='jabber:server'/>
     <xs:import namespace='urn:ietf:params:xml:ns:xmpp-sasl'/>
     <xs:import namespace='urn:ietf:params:xml:ns:xmpp-streams'/>
     <xs:import namespace='urn:ietf:params:xml:ns:xmpp-tls'/>

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

    <xs:element name='features'>
      <xs:complexType>
        <xs:sequence>
          <xs:any namespace='##other'
                  minOccurs='0'
                  maxOccurs='unbounded'
                  processContents='lax'/>
        </xs:sequence>
      </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:any     namespace='##other'
                      minOccurs='0'
                      maxOccurs='1'
                      processContents='lax'/>
        </xs:sequence>
      </xs:complexType>
    </xs:element>

  </xs:schema>
```

A.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='not-well-formed' type='empty'/>
     <xs:element name='policy-violation' type='empty'/>
     <xs:element name='remote-connection-failed' type='empty'/>
     <xs:element name='reset' 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: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='not-well-formed'/>
         <xs:element ref='policy-violation'/>
         <xs:element ref='remote-connection-failed'/>
         <xs:element ref='reset'/>
         <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: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>
```

A.3. STARTTLS 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:choice minOccurs='0' maxOccurs='1'>
           <xs:element name='required' type='empty'/>
         </xs:choice>
       </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>
```

A.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'
                       minOccurs='1'
                       maxOccurs='unbounded'
                       type='xs:NMTOKEN'/>
           <xs:any namespace='##other'
                   minOccurs='0'
                   maxOccurs='unbounded'
                   processContents='lax'/>
         </xs:sequence>
       </xs:complexType>
     </xs:element>

     <xs:element name='abort' type='empty'/>
     <xs:element name='auth'>
       <xs:complexType>
         <xs:simpleContent>
           <xs:extension base='xs:string'>
             <xs:attribute name='mechanism'
                           type='xs:NMTOKEN'
                           use='required'/>
           </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='success' type='xs:string'/>

     <xs:element name='failure'>
       <xs:complexType>
         <xs:sequence>
           <xs:choice minOccurs='0'>
             <xs:element name='aborted' type='empty'/>
             <xs:element name='account-disabled' type='empty'/>
             <xs:element name='credentials-expired' type='empty'/>
             <xs:element name='encryption-required' 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='malformed-request' 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:element ref='text' minOccurs='0' maxOccurs='1'/>
         </xs:sequence>
       </xs:complexType>
     </xs:element>

     <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>
```

A.5. Client Namespace

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

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

     <xs:import
         namespace='urn:ietf:params:xml:ns:xmpp-stanzas'/>

     <xs:element name='message'>
        <xs:complexType>
           <xs:sequence>
             <xs:choice minOccurs='0' maxOccurs='unbounded'>
               <xs:element ref='subject'/>
               <xs:element ref='body'/>
               <xs:element ref='thread'/>
             </xs:choice>
             <xs:any     namespace='##other'
                         minOccurs='0'
                         maxOccurs='unbounded'
                         processContents='lax'/>
             <xs:element ref='error'
                         minOccurs='0'/>
           </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='type'
                         use='optional'
                         default='normal'>
```

```
            <xs:simpleType>
              <xs:restriction base='xs:NMTOKEN'>
                <xs:enumeration value='chat'/>
                <xs:enumeration value='error'/>
                <xs:enumeration value='groupchat'/>
                <xs:enumeration value='headline'/>
                <xs:enumeration value='normal'/>
              </xs:restriction>
            </xs:simpleType>
          </xs:attribute>
          <xs:attribute ref='xml:lang' use='optional'/>
       </xs:complexType>
    </xs:element>

    <xs:element name='body'>
      <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:element name='subject'>
      <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:element name='thread'>
      <xs:complexType>
        <xs:simpleContent>
          <xs:extension base='xs:NMTOKEN'>
            <xs:attribute name='parent'
                          type='xs:NMTOKEN'
                          use='optional'/>
          </xs:extension>
        </xs:simpleContent>
      </xs:complexType>
    </xs:element>
```

```
     <xs:element name='presence'>
       <xs:complexType>
         <xs:sequence>
           <xs:choice minOccurs='0' maxOccurs='unbounded'>
             <xs:element ref='show'/>
             <xs:element ref='status'/>
             <xs:element ref='priority'/>
           </xs:choice>
           <xs:any     namespace='##other'
                       minOccurs='0'
                       maxOccurs='unbounded'
                       processContents='lax'/>
           <xs:element ref='error'
                       minOccurs='0'/>
         </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='type' use='optional'>
           <xs:simpleType>
             <xs:restriction base='xs:NMTOKEN'>
               <xs:enumeration value='error'/>
               <xs:enumeration value='probe'/>
               <xs:enumeration value='subscribe'/>
               <xs:enumeration value='subscribed'/>
               <xs:enumeration value='unavailable'/>
               <xs:enumeration value='unsubscribe'/>
               <xs:enumeration value='unsubscribed'/>
             </xs:restriction>
           </xs:simpleType>
         </xs:attribute>
         <xs:attribute ref='xml:lang' use='optional'/>
       </xs:complexType>
     </xs:element>
     <xs:element name='show'>
       <xs:simpleType>
         <xs:restriction base='xs:NMTOKEN'>
           <xs:enumeration value='away'/>
           <xs:enumeration value='chat'/>
           <xs:enumeration value='dnd'/>
           <xs:enumeration value='xa'/>
         </xs:restriction>
       </xs:simpleType>
     </xs:element>

     <xs:element name='status'>
       <xs:complexType>
         <xs:simpleContent>
           <xs:extension base='string1024'>
             <xs:attribute ref='xml:lang' use='optional'/>
           </xs:extension>
         </xs:simpleContent>
       </xs:complexType>
     </xs:element>

     <xs:simpleType name='string1024'>
       <xs:restriction base='xs:string'>
         <xs:minLength value='1'/>
         <xs:maxLength value='1024'/>
       </xs:restriction>
     </xs:simpleType>

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

     <xs:element name='iq'>
       <xs:complexType>
         <xs:sequence>
           <xs:any     namespace='##other'
                       minOccurs='0'
                       maxOccurs='1'
                       processContents='lax'/>
           <xs:element ref='error'
                       minOccurs='0'/>
         </xs:sequence>
         <xs:attribute name='from'
                       type='xs:string'
                       use='optional'/>
         <xs:attribute name='id'
                       type='xs:NMTOKEN'
                       use='required'/>
```

```
        <xs:attribute name='to'
                      type='xs:string'
                      use='optional'/>
        <xs:attribute name='type' use='required'>
          <xs:simpleType>
            <xs:restriction base='xs:NMTOKEN'>
              <xs:enumeration value='error'/>
              <xs:enumeration value='get'/>
              <xs:enumeration value='result'/>
              <xs:enumeration value='set'/>
            </xs:restriction>
          </xs:simpleType>
        </xs:attribute>
        <xs:attribute ref='xml:lang' use='optional'/>
      </xs:complexType>
    </xs:element>

    <xs:element name='error'>
      <xs:complexType>
        <xs:sequence xmlns:err='urn:ietf:params:xml:ns:xmpp-stanzas'>
          <xs:group ref='err:stanzaErrorGroup'/>
          <xs:element ref='err:text'
                      minOccurs='0'/>
        </xs:sequence>
        <xs:attribute name='by'
                      type='xs:string'
                      use='optional'/>
        <xs:attribute name='type' use='required'>
          <xs:simpleType>
            <xs:restriction base='xs:NMTOKEN'>
              <xs:enumeration value='auth'/>
              <xs:enumeration value='cancel'/>
              <xs:enumeration value='continue'/>
              <xs:enumeration value='modify'/>
              <xs:enumeration value='wait'/>
            </xs:restriction>
          </xs:simpleType>
        </xs:attribute>
      </xs:complexType>
    </xs:element>

  </xs:schema>
```

A.6. Server Namespace

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

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

     <xs:import
         namespace='urn:ietf:params:xml:ns:xmpp-stanzas'/>

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

```
            </xs:simpleType>
          </xs:attribute>
          <xs:attribute ref='xml:lang' use='optional'/>
       </xs:complexType>
    </xs:element>

    <xs:element name='body'>
      <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:element name='subject'>
      <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:element name='thread'>
      <xs:complexType>
        <xs:simpleContent>
          <xs:extension base='xs:NMTOKEN'>
            <xs:attribute name='parent'
                          type='xs:NMTOKEN'
                          use='optional'/>
          </xs:extension>
        </xs:simpleContent>
      </xs:complexType>
    </xs:element>
```

```
     <xs:element name='subject'>
       <xs:complexType>
         <xs:simpleContent>
           <xs:extension base='xs:NMTOKEN'>
             <xs:attribute name='parent'
                           type='xs:NMTOKEN'
                           use='optional'/>
           </xs:extension>
         </xs:simpleContent>
       </xs:complexType>
     </xs:element>

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

```
        </xs:attribute>
        <xs:attribute ref='xml:lang' use='optional'/>
      </xs:complexType>
    </xs:element>

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

    <xs:element name='status'>
      <xs:complexType>
        <xs:simpleContent>
          <xs:extension base='string1024'>
            <xs:attribute ref='xml:lang' use='optional'/>
          </xs:extension>
        </xs:simpleContent>
      </xs:complexType>
    </xs:element>

    <xs:simpleType name='string1024'>
      <xs:restriction base='xs:string'>
        <xs:minLength value='1'/>
        <xs:maxLength value='1024'/>
      </xs:restriction>
    </xs:simpleType>

    <xs:element name='priority' type='xs:byte' default='0'/>

    <xs:element name='iq'>
      <xs:complexType>
        <xs:sequence>
          <xs:any namespace='##other'
                  minOccurs='0'
                  maxOccurs='1'
                  processContents='lax'/>
          <xs:element ref='error'
                      minOccurs='0'/>
        </xs:sequence>
        <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='required'>
          <xs:simpleType>
            <xs:restriction base='xs:NMTOKEN'>
              <xs:enumeration value='error'/>
              <xs:enumeration value='get'/>
              <xs:enumeration value='result'/>
              <xs:enumeration value='set'/>
            </xs:restriction>
          </xs:simpleType>
        </xs:attribute>
        <xs:attribute ref='xml:lang' use='optional'/>
      </xs:complexType>
    </xs:element>

    <xs:element name='error'>
      <xs:complexType>
        <xs:sequence xmlns:err='urn:ietf:params:xml:ns:xmpp-stanzas'>
          <xs:group ref='err:stanzaErrorGroup'/>
          <xs:element ref='err:text'
                      minOccurs='0'/>
        </xs:sequence>
        <xs:attribute name='by'
                      type='xs:string'
                      use='optional'/>
        <xs:attribute name='type' use='required'>
          <xs:simpleType>
            <xs:restriction base='xs:NMTOKEN'>
              <xs:enumeration value='auth'/>
              <xs:enumeration value='cancel'/>
              <xs:enumeration value='continue'/>
              <xs:enumeration value='modify'/>
              <xs:enumeration value='wait'/>
            </xs:restriction>
          </xs:simpleType>
        </xs:attribute>
      </xs:complexType>
    </xs:element>

  </xs:schema>
```

A.7. 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>
           <xs:element name='resource' type='resourceType'/>
           <xs:element name='jid' type='fullJIDType'/>
         </xs:choice>
       </xs:complexType>
     </xs:element>

     <xs:simpleType name='fullJIDType'>
       <xs:restriction base='xs:string'>
         <xs:minLength value='8'/>
         <xs:maxLength value='3071'/>
       </xs:restriction>
     </xs:simpleType>

     <xs:simpleType name='resourceType'>
       <xs:restriction base='xs:string'>
         <xs:minLength value='1'/>
         <xs:maxLength value='1023'/>
       </xs:restriction>
     </xs:simpleType>

   </xs:schema>
```

A.8. Stanza Error Namespace

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

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

     <xs:element name='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='not-authorized' type='empty'/>
     <xs:element name='policy-violation' 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-authorized'/>
         <xs:element ref='not-allowed'/>
         <xs:element ref='policy-violation'/>
         <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 B. Contact Addresses

Consistent with [MAILBOXES], organization that offer XMPP services are encouraged to provide an Internet mailbox of "XMPP" for inquiries related to that service, where the host portion of the resulting mailto URI is the organization's domain, not the domain of the XMPP service itself (e.g., the XMPP service might be offered at im.example.com but the Internet mailbox would be <xmpp@example.com>).

Appendix C. Account Provisioning

Account provisioning is out of scope for this specification. Possible methods for account provisioning include account creation by a server administrator and in-band account registration using the 'jabber:iq:register' namespace as documented in [XEP-0077]. An XMPP server implementation or administrative function MUST ensure that any JID assigned during account provisioning (including localpart, domainpart, resourcepart, and separator characters) conforms to the canonical format for XMPP addresses defined in [XMPP-ADDR].

Appendix D. Differences from RFC 3920

Based on consensus derived from implementation and deployment experience as well as formal interoperability testing, the following substantive modifications were made from RFC 3920 (in addition to numerous changes of an editorial nature).

o Moved specification of the XMPP address format to a separate document.

- Recommended or mandated use of the 'from' and 'to' attributes on stream headers.

- More fully specified the stream closing handshake.

- Specified the recommended stream reconnection algorithm.

- Changed the name of the <xml-not-well-formed/> stream error condition to <not-well-formed/> for compliance with the XML specification.

- Removed the unnecessary and unused <invalid-id/> stream error (see RFC 3920 for historical documentation).

- Specified return of the <restricted-xml/> stream error in response to receipt of prohibited XML features.

- More completely specified the format and handling of the <see-other-host/> stream error, including consistency with RFC 3986 and RFC 5952 with regard to IPv6 addresses (e.g., enclosing the IPv6 address in square brackets '[' and ']').

- Specified that the SASL SCRAM mechanism is a mandatory-to-implement technology for client-to-server streams.

- Specified that TLS plus the SASL PLAIN mechanism is a mandatory-to-implement technology for client-to-server streams.

- Specified that support for the SASL EXTERNAL mechanism is required for servers but only recommended for clients (since end-user X.509 certificates are difficult to obtain and not yet widely deployed).

- Removed the hard two-connection rule for server-to-server streams.

- More clearly specified the certificate profile for both public key certificates and issuer certificates.

- Added the <reset/> stream error (Section 4.9.3.16) condition to handle expired/revoked certificates or the addition of security-critical features to an existing stream.

- Added the <account-disabled/>, <credentials-expired/>, <encryption-required/>, and <malformed-request/> SASL error conditions to handle error flows mistakenly left out of RFC 3920 or discussed in RFC 4422 but not in RFC 2222.

- Removed the unused <payment-required/> stanza error.

- Removed the unnecessary requirement for escaping of characters that map to certain predefined entities, since they do not need to be escaped in XML.

- Clarified the process of DNS SRV lookups and fallbacks.

- Clarified the handling of SASL security layers.

- Clarified that a SASL simple user name is the localpart, not the bare JID.

- Clarified the stream negotiation process and associated flow chart.

- Clarified the handling of stream features.

- Added a 'by' attribute to the <error/> element for stanza errors so that the entity that has detected the error can include its JID for diagnostic or tracking purposes.

- Clarified the handling of data that violates the well-formedness definitions for XML 1.0 and XML namespaces.

- Specified the security considerations in more detail, especially with regard to presence leaks and denial-of-service attacks.

- Moved documentation of the Server Dialback protocol from this specification to a separate specification maintained by the XMPP Standards Foundation.

Appendix E. Acknowledgements

This document is an update to, and derived from, RFC 3920. This document would have been impossible without the work of the contributors and commenters acknowledged there.

Hundreds of people have provided implementation feedback, bug reports, requests for clarification, and suggestions for improvement since publication of RFC 3920. Although the document editor has endeavored to address all such feedback, he is solely responsible for any remaining errors and ambiguities.

Special thanks are due to Kevin Smith, Matthew Wild, Dave Cridland, Philipp Hancke, Waqas Hussain, Florian Zeitz, Ben Campbell, Jehan Pages, Paul Aurich, Justin Karneges, Kurt Zeilenga, Simon Josefsson, Ralph Meijer, Curtis King, and others for their comments during Working Group Last Call.

Thanks also to Yaron Sheffer and Elwyn Davies for their reviews on behalf of the Security Directorate and the General Area Review Team, respectively.

The Working Group chairs were Ben Campbell and Joe Hildebrand. The responsible Area Director was Gonzalo Camarillo.

Author's Address

Peter Saint-Andre
Cisco
1899 Wyknoop Street, Suite 600
Denver, CO 80202
USA

Phone: +1-303-308-3282
EMail: psaintan@cisco.com