Network Working Group
Internet-Draft
Expires: December 14, 2000

J. Miller
The Jabber.org Project

June 15, 2000

Jabber
jabber

draft-miller-impp-jabber-00.txt

Status of this Memo

Copyright Notice

Abstract

This memo provides a comprehensive description of the Jabber architecture, design principles, and protocol.

Table of Contents

1. Introduction

1.1 What is Jabber?

At the core, Jabber is an API to provide instant messaging and presence functionality independent of data exchanged between entities. The primary use of Jabber is to give existing applications instant connectivity through messaging and presence features, contact list capabilities, and back-end services that transparently enrich the available functionality.

Essentially, Jabber defines an abstraction layer utilizing XML[1] to encode the common essential data types. This abstraction layer is managed by an intelligent server which routes data between the client APIs and the backend services that translate data from remote networks or protocols. By using this compatible abstraction layer, Jabber can provide many aspects of an Instant Messaging (IM) and/or Presence service in a simplified and uniform way.

1.2 Background and History

Jabber began in early 1998 as an open source project to both enable and ease the construction of compatible IM clients. The intention was that a client would only have to understand simple standard XML data types for messages and presence, and be able to focus on being an IM client, not on the complexity of dealing with the various IM networks. To avoid requiring the overhead of a typical "loadable module" API, the XML abstraction layer was made available to the client via a standard TCP[2] socket.

The chosen model is essentially that of a typical client-server, where the clients utilizing the API access it as a server via TCP. Certain server components have a very high frequency of upgrades, based on the adapting protocols with which they translate. The server was also then shared on a single host to a group of users to reduce the maintenance requirements for the client API. This also significantly reduces the deployment overhead for clients, since they simply need a single TCP socket to the server.

1.3 Evolution

As Jabber was developed and the architecture evolved, it became clear that a simple and flexible means of assigning identity to the various components was required. This started with clients, which are required to have a unique identity and be authorized to access the server API. The server was also assigned a unique identity, as were the individual components on the server side.

Due to the role that the server plays with respect to the clients,

and and due to the need to have a clear and established means of identitfying individual entities, the natural evolution of the server was to simply become an intelligent router. Not only does the server manage and route the XML data between the client API and the abstraction layer, but it also routes data directly between clients acting as an independent IM server for all clients. By assigning server identity as a hostname, servers are also able to route data directly to each other and their connected clients, thereby creating a new, inherently open IM network out of all the individual installed servers and clients.

Additionally, to simplify clients as well as to add functionality to the server, the server began to act as an XML repository by storing XML for the client. This enables the server to store the contact list (roster), offline messages, profile information such as a XML vCard[3], or custom client preferences and data (bookmarks, settings, etc.).

1.4 Today

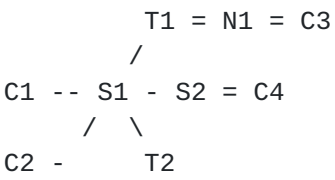
After two years of full-time development by a large group of individuals around the world:

- The abstraction layer and all APIs are fully implemented.
- The XML data types are well defined (Appendix A).
- Clients for all major platforms and environments are available or being developed. See http://jabbercentral.com/clients/ for a database of popular clients.
- The GPL[4]/LGPL[5] licensed Jabber Server[14] is fully functional.
- Numerous working server-side components (transports) exist or are in public development for popular services such as ICQ, AOL IM, Microsoft Messenger, Yahoo! Messenger, IRC, SMTP, RSS (news headlines), and more.

See http://jabber.org/ for more information.

2. Architecture Overview

Connection Map



Above is a map of the typical connections within the Jabber Architecture.

- "-" represents the Jabber XML Protocol.
- "=" represents any other protocol.
- C1, C2 - Jabber Clients.
- C3 - Client on another IM Network.
- C4 - Client using an alternate protocol to access the Jabber Server.
- S1 - a Jabber Server.
- T1 - Transport, translating between the Jabber XML Protocol and the protocol used on another IM Network.
- T2 - Transport providing other real-time data to Jabber, such as log notifications or headline news feeds.
- N1 - third party IM Network.

2.1 Server

The Server acts as an intelligent abstraction layer, managing the authorized client connections and connections to other Servers and Transports. All of the XML data is routed to the appropriate entities, or handled directly via internal modules on behalf of an entity. Servers can also utilize their own protocol to provide access to their clients.

2.2 Client

Clients may directly connect to the Server and use the XML Protocol to take full advantage of the functionality available. A client may also be customized to only operate with one server, or use an

alternate protocol to access a custom server. Clients of alternate IM Networks are also part of the Architecture, made accessable via a Transport to that IM Network.

2.3 Transport

A Tranport is a special-purpose Server. The primary function of a Transport is to translate the Jabber XML protocol to the protocol of a third party IM Network as well as translate the return data back to XML. Transports may also function to provide access to additional back-end functionality such as a real-time alert system, device presence, pager delivery, language translation or other custom needs.

3. Entity Identification

3.1 Three Tier

The basic concept behind all identity within Jabber is a three-tier structure: the host, node, and resource. This allows a host to manage its nodes, and allows each node to have independent addressable resources. The most common use of this structure is as a server, user, and connection identifier.

3.1.1 Host (server)

The basic required component of every ID is a Host. The Host is a standard DNS hostname and not case sensitive.

3.1.2 Node (user)

Each Host can be addressed with individual Nodes, or users. Each user is specific to the Host it is associated with, similar to email. Usernames are restricted to 255 characters, and the following ASCII characters are invalid: any character with a decimal value less than 33 or in the following set [:@<>'"&]. Case is preserved, but not used when comparing/matching. A Node address looks similar to email: node@host. Node addresses are intended to be human readable/usable in the same manner that email addresses are used today.

3.1.3 Resource

Resources are specific to a Node. All characters are allowed and there are no restrictions. Resource addresses are similar to the path part of a URL. Resources are used to address specific connections (since a Jabber server allows a user to be connected from multiple resources simultaneously), devices, or inboxes. An example Resource address: node@host/resource. Resource addresses are intended to be hidden from a user and only used at the software/protocol level.

3.2 URI

A Jabber identifier conforms to RFC 2396[6], "Uniform Resource Identifiers (URI): Generic Syntax" by prepending a "jabber://" to any address. When compared to other URIs, it acts like a hybrid mailto: and http:// URI, offering both a user identity and an optional path/resource specification.

Originally the abstraction layer used a Jabber URI within the protocol, but this was dropped until that use could undergo further review. Some of the reasoning was that the protocol was not capable

of handling any other URI, in the sense that SMTP does not directly handle mailto: URIs within the protocol. Also, there was a tendency to want to create proprietary custom URIs (such as 'icq:' or 'talk:') within the protocol. Enabling the abstraction layer to use full URIs directly was deferred until such a use could be standardized and agreed upon.

4. XML

4.1 Overview

XML[1] is used to define the common basic data types: message and presence. Essentially, XML is the core enabling technology within the abstraction layer, providing a common language with which everything can communicate. XML allows for painless growth and expansion of the basic data types and almost infinite customization and extensibility anywhere within the data. Many solutions already exist for handling and parsing XML, and the XML Industry has invested significant time in understanding the technology and ensuring full internationalization.

4.2 Namespaces

XML Namespaces[7] are used within all Jabber XML to create strict boundaries of data ownership. The basic function of namespaces is to separate different vocabularies of XML elements that are structurally mixed together. By ensuring that Jabber's XML is namespace-aware, it allows any XML defined by anyone to be structurally mixed with any data element within the protocol. This feature is relied upon frequently within the protocol to separate the XML that is processed by different components.

4.3 Validation

Provided with this specification is a reference DTD for Jabber. It is an important note that the Jabber server is not responsible for validating the XML elements forwarded to a new user - an implementation may choose to provide only validated data elements but is not required to.

Clients should not rely on the ability to send data which does not conform to the DTD, and should handle any non-conformant elements or attributes on the incoming stream by ignoring them.

4.4 <message> element

Messages are most similar in function to an email message.

4.4.1 Attributes

- to - Specifies to whom the message is intended to be delivered to.
- from - Specifies the sender of the message. Server set to prevent spoofing.

- type - Used to express the context as to what format the message should be displayed in. If no type is set, clients default to a type="normal". Values include:
  - normal - Single message dialog, similar to original ICQ messages.
  - chat - Traditional two-way chat similar to AIM or IRC CTCP Chat.
  - groupchat - Group interface similar to an IRC channel with multiple participants.
  - headline - Ticker or active list of items (news, stock market).
  - error - See the error element (Appendix C).

4.4.2 Children

- body - Contains the textual contents of the message for user display. No attributes.
- subject - Contains the subject of the message. Similar to an email subject. No attributes.
- thread - A random string generated by the originating client and copied back in replies. Used for tracking a conversation thread.
- error - See the error element (Appendix C).

4.4.3 Examples

The following examples have been server processed and contain the 'from' attribute.

A simple message:

```
<message to="horatio@denmark" from="hamlet@denmark">
  <body>Angels and Ministers of Grace, defend us!</body>
</message>
```

Complete chat message:

```
<message to="hamlet@denmark" from="horatio@denmark" type="chat">
  <subject>Plotting</subject>
  <body>Here, sweet lord, at your service.</body>
  <thread>100052</thread>
</message>
```

4.5 <presence> element

Presence is used to express the current status from one entity to a group of entities. The characteristics of presence are most similar to that of a network game where the data travels from one host to the server, and then broadcast to the other hosts participating in that game.

4.5.1 Attributes

- to - Specifies for whom the presence is bound. If none is specified, the server receives the presence.
- from - Whom the presence is from. Server set to prevent spoofing.
- id - A unique identifier for the presence. Sender of the presence sets this attribute.
- type - Describes the type of presence. No 'type' attribute implies "available" presence or current status. Allowable types include:
  - unavailable - Signals that the user is no longer available.
  - subscribe - An attempt to subscribe to the recipient's presence.
  - subscribed - The sender has enabled the recipient to receive their presence.
  - unsubscribe - An unsubscription request. The server handles the actual unsubscription, but clients receive a presence element for notification reasons.
  - unsubscribed - The subscription has been cancelled.
  - probe - A server-to-server query to request an entity's current presence.

4.5.2 Children

- show - Describes a user's exact availability. Must be one of:
  - away - User is away from the client software temporarily.
  - chat - User is free to chat.
  - xa - User is away for an extended period (eXtended Away).
  - dnd - User does not wish to be disturbed (Do Not Disturb).
- status - Custom availability message. Used in conjunction with the show element to give a detailed description of availability.

4.5.3 Examples

Initial presence sent to server upon login to express default availability:

```
<presence/>
```

Full-blown presence:

```
<presence from="hamlet@denmark">
  <show>xa</show>
  <status>Gone to England</status>
</presence>
```

4.6 <iq> element

Info/Query, or IQ, is a simple structured conversation wrapper. Just as HTTP is a request-response medium, IQ enables an entity to make a request and receive a response from another entity. The actual content of the request and response is defined by another namespace within the IQ.

4.6.1 Attributes

- to - Specifies for whom the IQ is bound.
- from - Specifies from whom the IQ is sent. Server set to prevent spoofing.
- id - A unique identifier for the IQ for tracking the query exchange. Sender of the IQ sets this attribute, which is returned in the response.
- type - The 'type' attribute has several preset values. Each

indicates a distinct step within an IQ conversation.

* get - Indicates that the current query is a question or search for information.

* set - This query contains data intended to set values or replace existing values.

* result - This is a successful response to a get/set query. The IQ usually contains no further information on a successful result.

* error - The query failed. See the error element (Appendix C).

4.6.2 Children

In the strictest terms, the iq element contains no children since it is a vessel for XML in another namespace. In operation, a query element is usually contained within the iq element as defined by its own separate namespace. See Appendix D.

4.6.3 Examples

The following examples are distinct parts of an IQ conversation for registration with jabber:iq:register (Appendix D.3) namespace.

Client request for registration information to a server service (service.denmark):

```
<iq type="get" id="1001" to="service.denmark">
  <query xmlns="jabber:iq:register"/>
</iq>
```

Server response with registration fields required:

```
<iq type="result" from="service.denmark" to="hamlet@denmark" id="1001">
  <query xmlns="jabber:iq:register">
    <instructions>Choose a username and password to register with this
server.</instructions>
    <name/>
    <email/>
    <password/>
    <key>106c0a7b5510f192a408a1d054150ed1065e255a</key>
  </query>
</iq>
```

Client request to register for an account:

```
<iq type="set" to="service.denmark" from="hamlet@denmark" id="1002">
  <query xmlns="jabber:iq:register">
    <name>hamlet</name>
    <email>hamlet@denmark</email>
    <password>gertrude</password>
    <key>106c0a7b5510f192a408a1d054150ed1065e255a</key>
  </query>
</iq>
```

Successful registration:

```
<iq type="result" from="service.denmark" to="hamlet@denmark" id="1002"/>
```

Failed registration:

```
<iq type="error" from="service.denmark" to="hamlet@denmark" id="1002"/>
  <error code="406">Not Acceptable</error>
</iq>
```

5. Client Access

5.1 TCP Socket

Clients connect to the server on TCP port 5222. The connection from the client to the server is persistent and maintains the presence state of the client after authentication. The server delivers all data to the client via this socket. If it is broken because of a network error, a client should reasonably attempt to reestablish the link with the server. Clients are never required to accept an incoming network connection, or establish a connection to any host other than the server.

5.2 Transport Layer

An XML Stream (Appendix B) is simply the direct association of an XML document with a single TCP socket, and is used as the default transport layer to move data between clients and servers. When the socket is opened, the first data sent is the opening root element of the XML document representing that stream. All XML data within the document is continually parsed as available via the socket. When a top-level element is closed, it can then be further processed by the application. This simple streaming transport layer reduces the complexity of implementing applications that use Jabber, and satisfies all the demands of the communication with the server.

5.3 Authentication

The client initially sends an IQ packet over the XML Stream to the server with the jabber:iq:auth namespace (Appendix D.1). This provides the credentials to access the server, which are either returned with an error or accepted. After accepting the credentials, the connection is then authorized to send and receive data.

5.4 Messages

The client sends a message with a valid 'to' attribute. The server processes the 'to' address and adds a 'from' address, then attempts to deliver to the recipient. Messages that fail for any reason are returned as an error. Messages may be delivered to the client from the server at any time, with an appropriate 'from' attribute set.

5.5 Presence

The current presence for the client is updated by sending presence to the server without a 'to' attribute. The server delivers this to the authorized recipients based on the subscription status as stored in the roster. The client will not receive presence from other entities until it has provided some form of available presence to

the server.

5.6 Roster

The client can send an IQ get with the jabber:iq:roster namespace (Appendix D.2) at any time, and will receive a result containing all the items that the server has stored for that user. Changes to any one item can be made by submitting that changed item via an IQ set in the same namespace.

Since Jabber allows multiple connections for a single user, the roster may change by one connection and that change needs to be updated to the other connections. At any time, the server may do a "Roster Push" by sending an IQ set to the client containing the new/updated item[s].

5.7 Presence Subscriptions

Presence information is made available only to approved entites. This approval is managed by a simple subscription system, using the presence element (Section 4.5). There are four distinct types of subscription requests:

- subscribe - A request to be approved for receiving all future presence changes
- unsubscribe - A request to no longer receive any presence changes
- subscribed - A notice that a subscription has been created and all future presence changes will be sent
- unsubscribed - A notice that a subscription has been removed, and no further presence changes will be sent

Request to subscribe to a user's presence:

```
<presence to="horatio@denmark" type="subscribe"/>
```

Response to a subscribe request:

```
<presence to="hamlet@denmark" type="subscribed"/>
```

6. Servers

The abstraction layer that Jabber provides includes the ability to individually address servers, and in fact relies on this ability for all external sources of data and functionality. This model then significantly expands the definition of a server to include intelligent agents, transports, device gateways, or any interactive data source. The additional expansion creates a need for flexibility in how server-side entities communicate.

The server to server communication happens independent of the client, and can use any protocol or communication means available. The server may choose to implement a loadable module interface to allow components (which might be addressed as a server) to interact with clients, or two servers may choose to tunnel their data via an alternate transport layer. In fact, the Jabber Server uses an alternate communication library, libetherx, which manages connections between the server and trusted transports in a more efficient model. Although any protocol can be used between servers, at least one common basic one must exist so that there can be some guarantee of interoperability.

6.1 Connections

Servers interconnect on TCP port 5269. There is no required state held between servers, so the connection is stateless and may be dropped at any time, but it is recommended that the connection persist based on the freqency of the interaction. Data is sent only over a locally originated connection to another server. All data from other servers is delivered via connections from those servers.

6.2 DNS

All connections are made directly to the IP address of the hostname in the Entity Identifier (Section 3). If the connection to this host fails, servers attempt an MX record lookup. A connection attempt is made to each found MX record. In order to not conflict with SMTP and be able to operate a Jabber server on a separate host, a list of MX records is processed in reverse priority (highest number first).

7. Security Considerations

7.1 SSL

Servers can optionally support normal SSL[8] connections for added security on port 5223 for client connections and 5270 for server connections.

7.2 Secure Identity and Encryption

Clients may optionally support signing and encrypting messages and presence by using PGP[15]/GnuPG[16].

The Jabber model specifically does NOT require trust in the remote servers or server-server trust. Although there may be benefits to a trusted server model (this issue is hotly debated), the direct client-client trust is already in use in email and allows those who desire a higher level of security to use it without requiring the significant increase in complexity throughout the architecture.

7.3 Client Connections

The IP address and method of access of clients is never made available, nor are any connections other than the original server connection required. This protects the client hosts from direct attack or identification by third parties, and allows the service provider to utilize an alternate protocol or provide another method of access to its clients.

7.4 Presence

Presence subscriptions are enforced by the user's server. Only the approved entities are able to discover a user's availability.

8. Scaling Considerations

8.1 TCP Sockets

A single, modern UNIX machine with proper tuning can handle 100,000 connections simultaneously. In c10k[10], Dan Kegel details how UNIX machines can be properly configured to handle large numbers of TCP connections.

8.2 Server Farms

By sending the 302 redirect error as response to an authorization attempt, clients can be intelligently re-routed to a group of servers. A large-scale Jabber Server deployment will function in a manner very similar to that of a web server farm, with each server handling a moderate load and adding server hardware as demand grows.

8.3 Client Optimizations

Custom clients can also make additional optimizations to reduce server load, such as idle auto-disconnect or local caching of server data that changes infrequently. These changes would be specific to the service requiring them and the clients supporting that service.

8.4 Server Restrictions

The server can apply limits to various aspects of the client functionality: roster size, offline message sizes, simultaneous session restrictions, rate limits, etc. These limits require no changes to clients and can be enforced on any server.

9. Extended Functionality

9.1 Services (Agents/Transports)

The Jabber Architecture is designed to transparently provide access to independent data sources, so it is inherently geared to be extended in this way. Any server-side entity can easily participate in the real-time generation and delivery of XML. These entities can expose simple data sources such as instant calendaring events or log notifications, or enable access to an existing group of individuals by translating to an alternate protocol. All the new functionality is immediately available to any application utilizing Jabber or any entity available via the abstraction layer.

9.2 MIME Transfers

All MIME objects, such as files, are transferred externally to the Jabber XML. MIME[11] is the dominant and successful method for transferring such data, and accessible in almost every environment. The Jabber XML Protocol is used to exchange http:// URLs between entities needing to exchange a MIME object. The jabber:x:oob can be inserted into messages and presence to express a MIME attachment. The jabber:iq:oob can be sent directly to another entity to express a single immediate MIME transfer. Recipient entities are then able to make the judgment whether they are able to handle the MIME objects, before incurring the cost of retreiving them.

9.3 XML Medium

As mentioned, the use of XML Namespaces allows any entity to insert any custom XML in an alternate namespace anywhere within the protocol. This flexibiliy is used frequently within clients and servers, and enables custom applications to utilize the access Jabber provides to instantly exchange structured data with each other.

Because Jabber is an XML medium, many of the new XML data formats can be immediately deployed on top of Jabber. This includes vCard XML[3], SyncML[17], XML/EDI[18], VoiceXML[19], and many more.

The server can also act as an XML repository. The Jabber Server supports this via the jabber:iq:private namespace, which can contain any other XML that is stored on the server for the client. This can be used to store client preferences, bookmarks, game states, address books, notes, and any other structured data.

A message with some custom XML included

```
<message to="horatio@denmark" from="hamlet@denmark">
  <body>Angels and Ministers of Grace, defend us!</body>
  <foo xmlns="http://www.foo.org/">
    <bar>ab<fb/>cd</bar>
  </foo>
</message>
```

9.4 Session Initialization

Users often use the existing presence and contact list to manually initiate other interactive software (e.g., "Hamlet, ready to start the game? I'm on and ready!"). By including custom XML or sending a URI, this could be automated within the client or application. Common uses are voice and video conferencing, games, collaborative tools, calendaring, and more.

9.5 Client Access

By utilizing the XML abstraction layer available as a client, additional gateways can be built to provide access to Jabber functionality for existing devices and software. Already, HTTP and IRC client gateways exist, with telnet and WAP[20] gateways under construction. In the future, devices with limited functionality or restricted access could use a custom gateway to access the messaging and presence features available on the server.

10. Examples

All data is sent over the TCP socket to denmark:5222.

SEND: indicates that the stream is going to the server while RECV: indicates that the stream is coming from the server to the client.

10.1 Minimal Client

A simple example:

```
SEND: <stream:stream
SEND:   to="denmark"
SEND:   xmlns="jabber:client"
SEND:   xmlns:stream="http://etherx.jabber.org/streams">

RECV: <stream:stream
RECV:   from="denmark"
RECV:   id="1001"
RECV:   xmlns="jabber:client"
RECV:   xmlns:stream="http://etherx.jabber.org/streams">

SEND:   <iq id="1001" type="set">
SEND:     <query xmlns="jabber:iq:auth">
SEND:       <username>hamlet</username>
SEND:       <password>gertrude</password>
SEND:       <resource>Castle</resource>
SEND:     </query>
SEND:   </iq>

RECV:   <iq id="1001" type="result"/>

SEND:   <message to="horatio@denmark">
SEND:     <body>The air bites shrewdly; it is very cold.</body>
SEND:   </message>

RECV:   <message to="hamlet@denmark" from="horatio@denmark>
RECV:     <body>It is nipping and an eager air.</body>
RECV:   </message>

SEND: </stream:stream>
RECV: </stream:stream>
```

10.2 Basic Client

A basic client converation using presence, rosters (iq) and messages:

```
SEND: <stream:stream
SEND:   to="denmark"
```

```
   SEND:   xmlns="jabber:client"
   SEND:   xmlns:stream="http://etherx.jabber.org/streams">

   RECV: <stream:stream
   RECV:   from="denmark"
   RECV:   id="1001"
   RECV:   xmlns="jabber:client"
   RECV:   xmlns:stream="http://etherx.jabber.org/streams">

   SEND:   <iq id="1001" type="set">
   SEND:     <query xmlns="jabber:iq:auth">
   SEND:       <username>hamlet</username>
   SEND:       <password>gertrude</password>
   SEND:       <resource>Castle</resource>
   SEND:     </query>
   SEND:   </iq>

   RECV:   <iq id="1001" type="result"/>

   SEND:   <iq id="1002" type="get">
   SEND:     <query xmlns="jabber:iq:roster"/>
   SEND:   </iq>

   RECV:   <iq id="1002" type="result">
   RECV:     <query xmlns="jabber:iq:roster>
   RECV:       <item jid="gertrude@denmark" name="Mother" subscription="both"/>
   RECV:       <item jid="polonius@denmark" subscription="both"/>
   RECV:       <item jid="horatio@denmark" name="Horatio" subscription="both">
   RECV:         <group>Friends</group>
   RECV:       </item>
   RECV:     </query>
   RECV:   </iq>

   SEND:   <presence/>

   RECV:   <presence from="polonius@denmark/work" to="hamlet@denmark">
   RECV:     <show>chat</show>
   RECV:   </presence>

   RECV:   <message from="polonius@denmark" to="hamlet@denmark">
   RECV:     <body>How does my good Lord Hamlet?</body>
   RECV:   </message>

   SEND:   <message to="polonius@denmark">
   SEND:     <body>Well, God-a-mercy.</body>
   SEND:   </message>

   RECV:   <message from="polonius@denmark" to="hamlet@denmark">
   RECV:     <body>Do you know me, my lord?</body>
```

```
RECV:   </message>

SEND: </stream:stream>
RECV: </stream:stream>
```

Note that only polonius@denmark is online at this time, thus no presence was received from other roster items.

10.3 Extending

A more advanced example that includes adding users to the roster, two new users coming online, a user changing status, and a client version request.

```
SEND: <stream:stream
SEND:   to="denmark"
SEND:   xmlns="jabber:client"
SEND:   xmlns:stream="http://etherx.jabber.org/streams">

RECV: <stream:stream
RECV:   from="denmark"
RECV:   id="1001"
RECV:   xmlns="jabber:client"
RECV:   xmlns:stream="http://etherx.jabber.org/streams">

SEND:   <iq id="1001" type="set">
SEND:     <query xmlns="jabber:iq:auth">
SEND:       <username>hamlet</username>
SEND:       <password>gertrude</password>
SEND:       <resource>Courtyard</resource>
SEND:     </query>
SEND:   </iq>

RECV:   <iq id="1001" type="result"/>

SEND:   <iq id="1002" type="get">
SEND:     <query xmlns="jabber:iq:roster"/>
SEND:   </iq>

RECV:   <iq id="1002" type="result">
RECV:     <query xmlns="jabber:iq:roster>
RECV:       <item jid="gertrude@denmark" name="Mother" subscription="both"/>
RECV:       <item jid="polonius@denmark" subscription="both"/>
RECV:       <item jid="horatio@denmark" name="Horatio" subscription="both">
RECV:         <group>Friends</group>
RECV:       </item>
RECV:     </query>
RECV:   </iq>
```

```
   SEND:   <presence/>

   RECV:   <presence from="polonius@denmark/work" to="hamlet@denmark">
   RECV:     <show>chat</show>
   RECV:   </presence>

   RECV:   <presence from="polonius@denmark/work" to="hamlet@denmark">
   RECV:     <show>away</show>
   RECV:     <status>Busy with project.</status>
   RECV:   </presence>

   RECV:   <presence from="rosencrantz@denmark" to="hamlet@denmark"
type="subscribe"/>
   RECV:   <presence from="guildenstern@denmark" to="hamlet@denmark"
type="subscribe"/>

   SEND:   <presence to="rosencrantz@denmark" type="subscribed"/>
   SEND:   <presence to="guildenstern@denmark" type="subscribed"/>

   RECV:   <iq type="set">
   RECV:     <query xmlns="jabber:iq:roster">
   RECV:       <item jid="rosencrantz@denmark" subscription="from"/>
   RECV:     </query>
   RECV:   </iq>

   RECV:   <iq type="set">
   RECV:     <query xmlns="jabber:iq:roster">
   RECV:       <item jid="guildenstern@denmark" subscription="from"/>
   RECV:     </query>
   RECV:   </iq>

   RECV:   <presence from="rosencrantz@denmark/trip" to="hamlet@denmark"/>
   RECV:   <presence from="guildenstern@denmark/palmpilot" to="hamlet@denmark"/>

   SEND:   <message to="polonius@denmark">
   SEND:     <body>Well, God-a-mercy.</body>
   SEND:   </message>

   RECV:   <message from="guildenstern@denmark" to="hamlet@denmark">
   RECV:     <body>My honoured lord!</body>
   RECV:   </message>

   RECV:   <message from="rosencrantz@denmark" to="hamlet@denmark">
   RECV:     <body>My most dear lord!</body>
   RECV:   </message>

   SEND:   <iq id="1003" type="get" to="rosencrantz@denmark/trip">
   SEND:     <query xmlns="jabber:iq:version"/>
   SEND:   </iq>
```

```
   RECV:   <iq type="result" from="rosencrantz@denmark/trip"
to="hamlet@denmark" id="1003">
```

```
   RECV:    <query xmlns="jabber:iq:version">
   RECV:      <name>Gabber</name>
   RECV:      <version>0.6.1</version>
   RECV:      <os>Debian GNU/Linux 2.2.16</os>
   RECV:    </query>
   RECV:  </iq>

   SEND: </stream:stream>
   RECV: </stream:stream>
```

11. IMPP and Interoperability Notes

11.1 Requirements Conformance

The implemented protocol presented in this memo is in near conformance to RFC 2778[12], "A Model for Presence and Instant Messaging" and RFC 2779[13], "Instant Messaging / Presence Protocol Requirements." Notable differences are outlined in the following section. It should be noted that the Jabber protocol has been in evolution for approximately two years as of the date of this memo, thus this protocol has not been designed in response to RFCs 2778 and 2779.

- RFC 2779, section 2.5 - Complete conformance with these requirements can be obtained by using a standard public key infrastructure such as GnuPG or PGP.

- RFC 2779, section 4.1, paragraph 10 - all MIME data is delivered via HTTP.

11.2 Interoperability

Jabber already provides complete interoperability, but at the cost of reverse engineering a 3rd party IM network protocol and operating a local translator to that protocol. The form of interoperability that Jabber offers is also restricted to the clients and requires the user to have a valid account on each IM network. It is the goal and desire of the Jabber development effort to increase the reach of the XML abstraction layer to overcome these limitations and reduce the reliance on local transports. The Jabber Development Team is ready and eager to interoperate directly at the server level with any other IM network and also willing to do much of the work to assist with this effort, as it reduces the complexity of the transport and adds significant new functionality to both sides.

Using MSN Messenger as a example, full interoperability can be achieved with a relatively simple addition to their IM network. The MSN Messenger service would directly translate the Jabber XML to and from the native format at the server level, as well as ensuring that the MSN Messenger client software would be able to address non-native users (it already utilizes a user@host based format, which may enable interoperability without any changes to client software). The actual details of the translation depend on the native data format and accessibility within the network of servers, but the XML is well defined and functionality across both networks is similiar enough for an almost direct mapping. This level of interoperability enables all Jabber and MSN Messenger users to communicate instantly and transparently without modifying parts of

either network.

References

[1] World Wide Web Consortium, "Extensible Markup Language (XML) 1.0", W3C xml, February 1998, <http://www.w3.org/TR/1998/REC-xml-19980210>.

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

[3] The Jabber.org Project, "vCard XML", March 2000, <http://protocol.jabber.org/vcard-temp/>.

[4] Free Software Foundation, "GNU General Public License", June 1991, <http://www.gnu.org/copyleft/gpl.html>.

[5] Free Software Foundation, "GNU Library General Public License", June 1991, <http://www.gnu.org/copyleft/lgpl.html>.

[6] Berners-Lee, T., Fielding, R.T. and L. Masinter, "Uniform Resource Identifiers (URI): Generic Syntax", RFC 2396, August 1998, <http://www.ietf.org/rfc/rfc2396.txt>.

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

[8] Freier, A., Karlton, P. and P. Kocher, "The SSL Protocol - Version 3.0", November 1996, <http://home.netscape.com/eng/ssl3/draft302.txt>.

[9] Fielding, R., Gettys, J., Mogul, J., Frystyk, H. and T. Berners-Lee, "Hypertext Transfer Protocol -- HTTP/1.1", RFC 2068, January 1997, <http://www.ietf.org/rfc/rfc2068.txt>.

[10] Kegel, D., "The C10K Problem", June 2000, <http://www.kegel.com/c10k.html>.

[11] Borenstein, N. and N. Freed, "MIME (Multipurpose Internet Mail Extensions) Part One: Mechanisms for Specifying and Describing the Format of Internet Message Bodies", RFC 1521, September 1993, <http://www.ietf.org/rfc/rfc1521.txt>.

[12] Day, M., Rosenberg, J. and H. Sugano, "A Model for Presence and Instant Messaging", RFC 2778, February 2000, <http://www.ietf.org/rfc/rfc2778.txt>.

[13] Day, M., Aggarwal, S., Mohr, G. and J. Vincent, "A Model for Presence and Instant Messaging", RFC 2779, February 2000,

<http://www.ietf.org/rfc/rfc2779.txt>.

[14] http://jabber.org

[15] http://www.pgp.com

[16] http://www.gnupg.org

[17] http://www.syncml.org

[18] http://www.xmledi-group.org/

[19] http://www.voicexml.org/

[20] http://www.wapforum.org/

Author's Address

Jeremie Miller
The Jabber.org Project
414 DeLong St.
Cascade, IA 52033
US

Phone: 319-852-3464
EMail: jeremie@jabber.org

Appendix A. The Jabber Protocol DTD

```
   <!--
       Jabber XML DTD
       By Julian "X-ViRGE" Missig
       julian@linuxpower.org
   -->

   <!--============ The Main Elements =============-->

   <!-- jabber, the root element -->
   <!-- note: the root element is not normally used due to the use of
xmlstreams and namespaces -->
   <!ELEMENT jabber ((presence | iq | message)*)>

   <!-- presence, a subelement of jabber -->

   <!ELEMENT presence ((status? | priority? | show?)*)>
   <!ATTLIST presence
     to          CDATA             #IMPLIED
     from             CDATA             #IMPLIED
     type        (subscribe | subscribed | unsubscribe | unsubscribed |
unavailable | probe) #IMPLIED
     >

   <!ELEMENT status (#PCDATA)>

   <!ELEMENT priority (#PCDATA)>

   <!ELEMENT show (#PCDATA)> <!-- standard options are: chat, away, xa and dnd
-->


   <!-- message, a subelement of jabber -->

   <!ELEMENT message ((body? | error* | subject? | thread?)*)>
   <!ATTLIST message
     to          CDATA             #IMPLIED
     from        CDATA             #IMPLIED
     id          ID
     type             (normal | error | chat | groupchat | headline) #IMPLIED
     >

   <!ELEMENT body (#PCDATA)>

   <!ELEMENT error (#PCDATA)>

   <!ATTLIST error
     code        CDATA        #IMPLIED
```

```
    >

  <!ELEMENT subject (#PCDATA)>
```

```
   <!ELEMENT thread (#PCDATA)>

   <!ELEMENT x (#PCDATA)>


   <!-- iq, a subelement of jabber -->

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

Appendix B. XML Streams DTD

```
<!ELEMENT stream (error?)>
<!ATTLIST stream
  to              CDATA       #REQUIRED
  from            CDATA       #IMPLIED
  id              CDATA       #IMPLIED
  xmlns           CDATA       #REQUIRED
  xmlns:stream    CDATA       #REQUIRED 'http://etherx.jabber.org/streams'
  >
<!ELEMENT error (PCDATA)>
```

Appendix C. <error> element

A standard error element is used for failed processing of messages and iq. This element is a child of the failed element.

C.1 Attributes

- code - a numerical error code corresponding to a specific error description. The numerical codes used are nearly synchronous with HTTP error codes:
  - 302 - Redirect
  - 400 - Bad Request
  - 401 - Unauthorized
  - 402 - Payment Required
  - 403 - Forbidden
  - 404 - Not Found
  - 405 - Not Allowed
  - 406 - Not Acceptable
  - 407 - Registration Required
  - 408 - Request Timeout
  - 500 - Internal Server Error
  - 501 - Not Implemented
  - 502 - Remote Server Error
  - 503 - Service Unavailable
  - 504 - Remote Server Timeout

C.2 Examples

Message error:

```
<message to="hamlet@denmark" from="horatio@denmark" type="error">
  <body>Angels and Ministers of Grace, defend us!</body>
  <error code="404">Not Found</error>
</message>
```

IQ Error:

```
<iq type="error" from="service.denmark" to="hamlet@denmark" id="1002">
  <query xmlns="jabber:iq:register">
    <name>hamlet</name>
    <email>hamlet@denmark</email>
    <password>gertrude</password>
    <key>106c0a7b5510f192a408a1d054150ed1065e255a</key>
  </query>
  <error code="502">Remote Server Error</error>
</iq>
```

Appendix D. Info/Query Namespaces

Numerous Info/Query (Section 4.6) namespaces have been implemented to faciliate exchange of information between Jabber entities. Namespaces currently implemented within the Jabber server include:

- Simple Client Authentication (jabber:iq:auth)
- Agent Properties (jabber:iq:agent)
- Registration Requests (jabber:iq:register)
- Roster (Contact List) Management (jabber:iq:roster)
- Available Agents List (jabber:iq:agents)
- Out Of Band Data (jabber:iq:oob)
- Client Time (jabber:iq:time)
- Client Version (jabber:iq:version)
- Temporary vCard (vcard-temp)

Note that the Temporary vCard namespace is being used until the vCard XML standard has been finalized.

The following subsections describe the three most fundamental extensions.

D.1 Simple Client Authentication - jabber:iq:auth

The jabber:iq:auth namespaces provides a simple mechanism for clients to authenticate and create a resource representing their connection to the server.

D.1.1 Children

- username - the unique user name for this user.
- password - the secret key or passphrase for the user account.
- digest - send a password in a SHA1 hash instead of clear text password.
- resource - unique value to represent current connection.

D.1.2 Examples

The following is a complete example of how a client authenticates with the server.

Client sends user information:

```
<iq type="set" id="1001">
  <query xmlns="jabber:iq:auth">
    <username>hamlet</username>
    <password>gertrude</password>
     <resource>Castle</resource>
  </query>
</iq>
```

Server confirms login:

```
<iq type="result" id="1001"/>
```

D.2 Roster (Contact List) Management - jabber:iq:roster

Provides a simple method for server-side contact list management. Upon connecting to the server, clients should request for the roster using jabber:iq:roster. Since the roster may not be desirable for all clients (e.g., cellular phone client), the client request of the roster is optional.

D.2.1 Children

- item - a specific roster item (contact) has the following attributes:
  - jid - the complete JID (Jabber ID) of the user that this item represents
  - subscription - the current status of the subscription related to this item. May have a value of:
    - none - no subscription.
    - from - this entity has a subscription to the user.
    - to - the user has a subscription to this entity.
    - both - subscription is both to and from.
    - remove - item is to be removed.
  - ask - the current status of a request to this item. May be one

of:

+ subscription - the user is asking this item for a subscription.

+ unsubscription - the user is asking this item for an unsubscription.

This element may contain one or more instances of the following element:

* group - contains a user-specified user group name.

D.2.2 Examples

Client request for current roster:

```
<iq type="get" id="1001">
  <query xmlns="jabber:iq:roster"/>
</iq>
```

Server response to client query:

```
<iq type="result" id="1001">
  <query xmlns="jabber:iq:roster">
    <item jid="claudius@denmark" name="Uncle Claudius" subscription="from">
      <group>Family</group>
    </item>
    <item jid="horatio@denmark" name="Horatio" subscription="both">
      <group>Friends</group>
    </item>
    <item jid="fortinbras@norway" name="Prince Fortinbras"
subscription="none" ask="subscribe"/>
  </query>
</iq>
```

Client adding new items and modifying an entry:

```
<iq type="set" id="1002">
  <query xmlns="jabber:iq:roster">
    <item name="Rosencrantz" jid="rosencrantz@denmark">
      <group>Visitors</group>
    </item>
    <item name="Guildenstern" jid="guildenstern@denmark">
      <group>Visitors</group>
    </item>
    <item jid="claudius@denmark" name="King Claudius">
      <group>Family</group>
      <group>Royalty</group>
    </item>
  </group>
</iq>
```

The server would then respond with the new roster information, plus an IQ result:

```
<iq type="set">
  <query xmlns="jabber:iq:roster">
    <item jid="rosencrantz@denmark" name="Rosencrantz">
      <group>Visitors</group>
    </item>
    <item jid="guildenstern@denmark" name="Guildenstern" >
      <group>Visitors</group>
    </item>
    <item jid="claudius@denmark" name="King Claudius">
      <group>Family</group>
       <group>Royalty</group>
    </item>
  </group>
</iq>
<iq type="result" id="1002"/>
```

D.3 Registration Request - jabber:iq:register

Through jabber:iq:register, clients can register with the Jabber server itself or with new services.

D.3.1 Children

Note that while numerous fields are available, only the ones returned by the server are required for registration.

- username

- password

- name
- email
- address
- city
- state
- zip
- phone
- url
- date
- misc
- text
- instructions - contains server provided instructions for registration.
- key - a unique key provided by the server, required for the entire registration process.

D.3.2 Examples

A complete example is provided in the IQ examples (Section 4.6.3).

Appendix E. X Namespaces

For sending information that does not require the IQ structure, the X namespace series has been implemented. Clients can use this type of namespace to send URLs, Roster (Contact List) items, Offline Options and other information. The following X namespaces have been implemented so far in the Jabber server:

- Delay Logging (jabber:x:delay)
- Out Of Band Data (File Transfers) (jabber:x:oob)
- Embedded Roster Items (jabber:x:roster)

E.1 Examples

Sending an embedded roster item to a user:

```
<message to="hamlet@denmark" from="horatio@denmark">
  <subject>Visitors</subject>
  <body>This message contains roster items.</body>
  <x xmlns="jabber:x:roster">
    <item jid="rosencrantz@denmark" name="Rosencrantz"><group>Visitors</
group></item>
    <item jid="guildenstern@denmark" name="Guildenstern"><group>Visitors</
group></item>
  </x>
</message>
```

Appendix F. Acknowledgments

While the entire Jabber.org team has been actively involved in the development of this protocol, the following individuals have significantly contributed:

Eliot Landrum

Thomas Muldowney

Thomas Charron

Julian Missig

Peter Millard

Full Copyright Statement

Acknowledgement

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