Gnutella Protocol Development

Network Working Group   T. Klingberg
Request for Comments: NNNN   R. Manfredi
Category: Informational   June 2002

Gnutella 0.6

Status of this Memo

This is a draft.

Copyright Notice

Rights of Free Implementation

The authors of the various proposals that make up this document grant the rights to anyone to freely implement those proposals. Gnutella is an Open Protocol, where the specifications are public and free of any patent.

Table of Contents

1 Introduction

1.1 Purpose

Gnutella is a decentralized peer-to-peer system. It allows the participants to share resources from their system for others to see and get, and locate resources shared by others on the network.

Resources can be anything: mappings to other resources, cryptographic keys, files of any type, meta-information on keyable resources, etc. However, the semantics for locating and handling resources other than plain files are not specified in this document.

Each participant launches a Gnutella program, which will seek out other Gnutella nodes to connect to. This set of connected nodes carries the Gnutella traffic, which is essentially made of queries, replies to those queries, and also other control messages to facilitate the discovery of other nodes.

Users interact with the nodes by supplying them with the list of resources they wish to share on the network, can enter searches for other's resources, will hopefully get results from those searches, and can then select those resources amongst the results: if those resources are files, for instance, they can download them. But one can imagine other types of resources that, once fetched, will bring more than their content value.

Resource data exchanges between nodes are negotiated using the standard HTTP protocol. The Gnutella network is only used to locate the nodes sharing those resources.

This document is intended for readers with a fair knowledge of network programming, but do not require any previous Gnutella experience. Still, other implementations of this protocol will give useful information about implementation techniques that is not included in this document. A list of Gnutella programs can be found at http://www.gnutelliums.com

1.2 Requirements

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in RFC 2119 [34].

1.2.1 The Gnutella Development Forum (the GDF)

The Gnutella Development Forum is a good place to find more Gnutella documentation, proposals about changes and extensions and to discuss Gnutella development with other developers. The message archive is also a good source for information about the protocol and its implementation. Some of the links in this document requires membership in the Gnutella Development Forum. Everyone is, of course, allowed to become a member. The GDF is located at http://groups.yahoo.com/group/the_gdf

There are many other forums for discussing Gnutella development as well.

1.3 Terminology

Servent	A program participating in the Gnutella network is called a servent. The words "peer", "node" and "host" have similar meanings, but refers to a network participant rather than a program. When a servent have a clear client or server role the words "client" or "server" may be used. The word "client" is

	sometimes used as a synonym for servent. This is a contraction of "SERVer" and "cliENT", Some other documents use the word "servant" instead of servent.
Message	Messages are the entity in which information is transmitted over the network. Sometimes the word "packet" is used with the same meaning. Some other documents use the word "descriptor"
GUID	Globally Unique IDentifier. This is a 16-byte long value made of random bytes, whose purpose it is to identify servents and messages. This identification is not a signature, just a way to identify network entities in a unique manner.

1.4 Extending the protocol

This document is the definition of the Gnutella 0.6 protocol. Servents MAY extend the protocol or even change parts of it (for example by compressing or encrypting the messages), but servents MUST always stay compatible with servents that follow this specification.

If a servent, for example, wants to compress the Gnutella messages, it MUST first make sure the remote host of a connection can decompress the stream (during handshake), and otherwise leave the messages uncompressed. Servents MAY chose not to accept a connection with a servent that does not support a feature, but MUST always make sure that the Gnutella network is not split into separate networks.

Separate networks for special purposes are, of course, allowed but then it is no longer the Gnutella network, but another network.

This protocol also allows for extensions inside many messages. Such extensions can pass through servents that do not know about the extension to reach servents that do.

2 Protocol Definition

The Gnutella protocol defines the way in which servents communicate over the network. It consists of a set of messages used for communicating data between servents and a set of rules governing the inter-servent exchange of messages. Currently, the following messages are defined:

Ping	Used to actively discover hosts on the network. A servent receiving a Ping message is expected to respond with one or more Pong messages.
Pong	The response to a Ping. Includes the address of a connected Gnutella servent, the listening port of that servent, and information regarding the amount of data it is making available to the network.
Query	The primary mechanism for searching the distributed network. A servent receiving a Query message will respond with a Query Hit if a match is found against its local data set.
QueryHit	The response to a Query. This message provides the recipient with enough information to acquire the data matching the corresponding Query.
Push	A mechanism that allows a firewalled servent to contribute file-based data to the network.
Bye	An optional message used to inform the remote host that you are closing the connection, and your reason for doing so.

2.1 Initiating a Connection

A Gnutella servent connects itself to the network by establishing a connection with another servent currently on the network. Techniques for finding the first host are described in Appendix 3. Once the first connection is established, the addresses of more hosts will be supplied over the network. The default Gnutella port is 6346, but servents MAY use any unused port. If the desired port is used (probably by another Gnutella servent) the servent SHOULD attempt to

```
listen on another port.  This listening port is advertised by the
servent through the Pong messages.

Techniques and rules for how to select what other Gnutella hosts to
connect to and when to accept connection requests can be found in
Appendix 4.

Once the address of another servent on the network is obtained, a
TCP/IP connection to the servent is created, and a handshaking
sequence is initiated. The client is the host initiating the
connection and the server is the host receiving it. "" refers
to ASCII character 13 (carriage return), and "" to 10 (new line).

   1. The client establishes a TCP connection with the server.
   2. The client sends "GNUTELLA CONNECT/0.6".
   3. The client sends all capability headers--except for
      vendor-specific headers--each terminated by "", with
      an extra "" at the end.
   4. The server responds with "GNUTELLA/0.6 200 ".
       SHOULD be "OK", but servents SHOULD just look for the
      "200" code.
   5. The server sends all its headers, in the same format as in (3).
   6. The client sends "GNUTELLA/0.6 200 OK, as in (4) if
      after parsing the server's headers, it still wishes to connect.
      Otherwise, it needs to reply with an error code and close the
      connection.
   7. The client sends any vendor-specific headers as needed, in the
      same format as (3).
   8. Both client and server send binary messages at will, using the
      information gained in (3) and (5).

All headers SHOULD be registered with the GDF database at
http://groups.yahoo.com/group/the_gdf/database?method=reportRows&tbl=9
(Requires GDF membership)

Headers follow the standards described in RFC822 and RFC2616.  Each
header is made of a field name, followed by a colon, and then the
value.  Each line ends with the  sequence, and the end of the
headers is marked by a single  line.  Each line normally
starts a new header, unless it begins with a space or an horizontal
tab (ASCII codes 32 and 9 in decimal, respectively), in which case it
continues the preceding header line.  The extra spaces and tabs may
be collapsed into a single space as far as the header value goes.
For instance:

    First-Field: this is the value of the first field
    Second-Field: this is the value
        of the
        second field


The header above is made of two fields, "First-Field" and "Second-
Field" whose values are respectively "this is the value of the first
field" and "this is the value of the second field" (leading spaces of
the continuation were collapsed).  Note that the leading space
between the ":" ending the field name and the start of the value
string does not count.

Multiple header lines with the same field name are identical to one
header line where all the values of the fields would be separated by
",". This means:

    Field: first
    Field: second

is strictly equivalent to saying:

    Field: first,second

In other words, order matters in that case.

Here is a sample interaction between a client and a server.  Data
sent from client to server is shown on the left; data sent from
server to client is shown on the right.

     Client                           Server
     -----------------------------------------------------------
     GNUTELLA CONNECT/0.6
     User-Agent: BearShare/1.0
     Pong-Caching: 0.1
     GGEP: 0.5
```

```
                                                GNUTELLA/0.6 200 OK
                                                User-Agent: BearShare/1.0
                                                Pong-Caching: 0.1
                                                GGEP: 0.5
                                                Private-Data: 5ef89a

      GNUTELLA/0.6 200 OK
      Private-Data: a04fce


      [binary messages]                [binary messages]
```

A few notes about the responses: first, the client (server) SHOULD disconnect if receiving any response other than "200" at step 4 (6). There is no need to define these error codes now. Second, servents SHOULD ignore higher version numbers in steps (2), (4), and (6). For example, it is perfectly legal for a future client to connect to a server and send "GNUTELLA CONNECT/0.7". The server SHOULD respond with "GNUTELLA/0.7 200 OK" if it supports the 0.7 protocol, or "GNUTELLA/0.6 200 OK" otherwise.

A few notes about the headers: servents SHOULD use standard HTTP headers whenever appropriate. For example, servents SHOULD use the standard "User-Agent" header rather than make up a "Servent-Vendor" header. However, it is perfectly legal to add new headers (e.g., "Query-Routing") when no appropriate HTTP header exists, as long as they follow HTTP syntax. Headers unknown to the servent MUST be ignored.

Some older servents will initiate the handshake by sending "GNUTELLA CONNECT/0.4". The server SHOULD then reply with "GNUTELLA OK" followed by binary messages, if it can accept the connection. Servents MAY retry using the 0.4 connect string if the 0.6 connection attempt were rejected. No handshaking headers can be used in 0.4 handshaking.

When rejecting a connection, a servent MUST, if possible, provide the remote host with a list of other Gnutella hosts, so it can try connecting to them. This SHOULD be done using the X-Try header.

An X-Try header can look like:

```
        X-Try:1.2.3.4:1234,3.4.5.6:3456
```

There MAY be a space after the colon and after each comma. There MAY be multiple X-Try headers in one header set. The header MAY end with an extra comma. The header MAY be formatted on several lines using continuations.

Each item in the X-Try header gives the IP address of a servent and its listening port number. This is sometimes referred to as being a "connection pong". If the server sending the X-Try implements Pong-Caching, then the connection pongs being sent must be fresh ones.

The normal status code for rejecting a connection because the servent is busy is "503 " followed by "Busy" or another description string.

2.2 Gnutella Messages

Once a servent has connected successfully to the network, it communicates with other servents by sending and receiving Gnutella protocol messages. Each message is preceded by a Message Header with the byte structure given below.

Note 1: One IP packet may contain several Gnutella messages, and one Gnutella message may be split up on multiple IP-packets. This means one can never assume a Gnutella message ends when the chunk of data read from the socket ends.

Note 2: All fields in the following structures are in little-endian byte order unless otherwise specified.

Note 3: All IP addresses in the following structures are in IPv4 format. For example, the IPv4 byte array

```
    0xD0     0x11     0x32     0x04
    byte 0   byte 1   byte 2   byte 3
```

represents the dotted address 208.17.50.4.

2.2.1 Message Header

The message header is 23 bytes divided into the following fields.

Bytes:	Description:
0-15	Message ID/GUID (Globally Unique ID)
16	Payload Type
17	TTL (Time To Live)
18	Hops
19-22	Payload Length

Message ID	A 16-byte string (GUID) uniquely identifying the message on the network. Servents SHOULD store all 1's (0xff) in byte 8 of the GUID. (Bytes are numbered 0-15, inclusive.) This serves to tag the GUID as being from a modern servent. Servents SHOULD initially store all 0's in byte 15 of the GUID. This is reserved for future use. The other bytes SHOULD have random values.
Payload Type	Indicates the type of message 0x00 = Ping 0x01 = Pong 0x02 = Bye 0x40 = Push 0x80 = Query 0x81 = Query Hit Other Gnutella messages can be used, but if so the servent MUST first make sure that the remote host supports this new message type. This can be done using handshaking headers.
TTL	Time To Live. The number of times the message will be forwarded by Gnutella servents before it is removed from the network. Each servent will decrement the TTL before passing it on to another servent. When the TTL reaches 0, the message will no longer be forwarded (and MUST not).
Hops	The number of times the message has been forwarded. As a message is passed from servent to servent, the TTL and Hops fields of the header must satisfy the following condition: TTL(0) = TTL(i) + Hops(i) Where TTL(i) and Hops(i) are the value of the TTL and Hops fields of the message, and TTL(0) is maximum number of hops a message will travel (usually 7).
Payload Length	The length of the message immediately following this header. The next message header is located exactly this number of bytes from the end of this header i.e. there are no gaps or pad bytes in the Gnutella data stream. Messages SHOULD NOT be larger than 4 kB.

The Payload Length field is the only reliable way for a servent to find the beginning of the next message in the input stream. Therefore, servents SHOULD rigorously validate the Payload Length field for each message received. If a servent becomes out of synch with its input stream, it SHOULD close the connection associated with the stream since the upstream servent is either generating, or forwarding, invalid messages.

Abuse of the TTL field in broadcasted messages (Query) will lead to an unnecessary amount of network traffic and poor network performance. Therefore, servents SHOULD carefully check the TTL fields of received query messages and lower them as necessary. Assuming the servent's maximum admissible Query message life is 7 hops, then if TTL + Hops > 7, TTL SHOULD be decreased so that TTL + Hops = 7. Broadcasted messages with very high TTL values (>15) SHOULD be dropped.

Immediately following the message header, is a payload consisting of one of the following messages.

2.2.2 Ping (0x00)

Ping messages MAY contain a GGEP extension block (see Section 2.3), but no other payload.

2.2.3 Pong (0x01)

Pong messages contains information about a Gnutella host. The message has the following fields

Bytes:	Description:
0-1	Port number. The port number on which the responding host can accept incoming connections.
2-5	IP Address. The IP address of the responding host. Note: This field is in big-endian format.
6-9	Number of shared files. The number of files that the servent with the given IP address and port is sharing on the network.
10-13	Number of kilobytes shared. The number of kilobytes of data that the servent with the given IP address and port is sharing on the network.
14-	OPTIONAL GGEP extension block. (see Section 2.3)

Pong messages are only sent in response to an incoming Ping message. It is valid for more than one Pong message to be sent in response to a single Ping message. This enables host caches to send cached servent address information in response to a Ping request.

The Message ID of a Pong message MUST be the Message ID of the Ping message it is sent in reply to.

The fields specifying the number of shared files and the number of kilobytes shared was intended to allow one to measure the amount of data available on the network. With a very large Gnutella network, and minimized Ping and Pong message traffic, this can no longer be done. Still, these fields SHOULD be filled out correctly.

2.2.4 Use of Ping and Pong messages

In early versions Gnutella, Ping messages were broadcasted over the network. Pong messages were then routed back to the originator of the Ping message the same way as Query Hits messages are routed (se section 2.2.7). That system consumed a lot of network bandwidth, so modern Gnutella servents cache Pong messages, or use other means of minimizing the bandwidth used by Ping and Pong messages.

There are different systems for handling Ping and Pong messages, but what they have in common is:

* When a Ping message is received (TTL>1 and it was at least one second since another Ping was received on that connection), a servent MUST, if possible, respond with a number of Pong Messages. These pongs MUST have the same message ID as the incoming ping, and a TTL no lower than the hops value of the ping. The number of pongs returned may vary, but 10 is a reasonable number. Servents that are able to accept incoming Gnutella SHOULD reply to these Ping messages.

* The pongs sent SHOULD have a good quality. That includes high probability that they are connectable and a good spread of hosts from across the network

* The bandwidth used by Ping and Pong messages SHOULD be minimized. Servents MUST never output very high quantities of Ping and Pong messages.

* An incoming Ping message with TTL = 1 and Hops = 0 or 1 is used to probe the remote host of a connection, and MUST always be replied to with a pong having information about the host who received the ping.

* An incoming Ping message with TTL = 2 and Hops = 0 is a "Crawler Ping" used to scan the network. It and SHOULD be replied to with pongs containing information about the host receiving the ping and all other hosts it is connected to. The information about neighbour nodes can be provided either by

creating pongs on their behalf, or by forwarding the ping to them, and forward the pongs returned to the crawler.

Servents fulfilling these requirements MUST provide a the header "Pong-Caching: 0.1" (or a higher number if a later version is used) during the handshake. That allows other nodes to know if pong caching in any form is supported. Note that this applies to servents do not really cache pong messages as well, as long as the rules above applies. Servents are strongly RECOMMENDED to follow the rules above, and provide the Pong-Caching header.

When storing or forwarding Pong messages, any GGEP payload SHOULD be included. When sending a Ping message, one cannot know if it will reach only the neighbour host, or many hosts on the network. It depends on what system for handling Ping and Pong messages other servents are using. Servents MUST NOT make assumptions of how far a Ping message (and its payload) will reach.

2.2.4.1 A simple pong caching scheme

This is one system for handling Ping and Pong messages. There are others available (see sect. 2.2.4.2), and any system that abides to the rules in sect. 2.2.4 is ok.

For each connection an array of Pong Messages are stored. 10 may be a good number. When a pong comes in, it overwrites the oldest stored pong in array of he connection the pong came from. The information that must be stored for each pong is:

* IP Address
* Port number
* Number of files shared
* Number of kilobytes shared
* GGEP extension block (if present)
* Hops value, i.e. how far away on the network the host using the stored address is

When a Ping message, called P, is received over connection C, and it has been at least one second since last time a ping was received over C, the servent will return a number of pongs (10 for example) from its stored pongs. The pongs will be pick from all connections except from C, since it would be no good sending pongs back where they came from. A servent should also return a pong with information about itself, if it can accept incoming connections.

The outgoing pong will have the same message ID as P, not the message ID it had when the pong was received. The Hops is set to the stored hops value + 1, and TTL so that TTL+Hops=7. If the TTL is less than P's Hops value, the current stored pong will not be sent. This also means that pongs whose Hops value already is 7 will not be propagated any further.

Exactly how to select which of the stored pongs to send in response to an incoming ping is up to each servent. A good idea is to pick pongs from different connections and with varying stored Hops values.

To keep the cache fresh, a ping (TTL=7, Hops=0) is sent over all connections at small interval (like every 3 seconds). This look like very often, but remember that the neighbour servents will just respond with pongs from its own cache. The short time ensures that pongs are always fresh. To neighbour hosts who has not indicated that they support pong caching (using the Pong-Caching handshaking header), one ping per minute might be a better number.

Incoming pings with TTL=1 and Hops=0 or 1 (see above section 2.2.4) is replied to with a single pong containing information about the local host. Pings with TTL=2 and Hops=0 are replied to with one pong about the local host, and one about each other host the local host is connected to. Information about the neighbour hosts is retrieved when a new connection is started by sending a TTL=1, Hops=0 ping and storing the pong returned. This can be done using handshaking headers instead.

The bandwidth used by this scheme is very limited. Assuming a ping is sent every 3 seconds and that 10 pongs are returned to every ping. Since a (without extensions) is 23 bytes and a pong (without extensions) is 37 bytes, the amount of bandwidth used per connection is (23+10*37)/3 = 131 bytes/sec/connection. If extensions are used in ping and/or pong messages, the bandwidth usage will increase, but will still be kept on an acceptable level. If the bandwidth usage must re decreased further, the interval between update pings could be increased.

2.2.4.2 Other pong caching schemes

A slightly more advanced scheme for pong caching is available at http://www.limewire.com/index.jsp/pingpong

A different, but compatible scheme can be found at http://groups.yahoo.com/group/the_gdf/files/Proposals/PONG/Variants/pingreduce.txt

Other schemes might have been created after this was written.

2.2.5 Query (0x80)

Since Query messages are broadcasted to many nodes, the total size of the message SHOULD not be larger than 256 bytes. Servents MAY drop Query messages larger that 256 bytes, and SHOULD drop Query messages larger than 4 kB.

A Query message has the following fields:

Bytes:	Description:
0-1	Minimum Speed. The minimum speed (in kb/second) of servents that should respond to this message. A servent receiving a Query message with a Minimum Speed field of n kb/s SHOULD only respond with a Query Hit if it is able to communicate at a speed >= n kb/s.
2-	Search Criteria. This field is terminated by a NUL (0x00). See section 2.2.7.3 for rules and information on how to interpret the Search Criteria
Rest	OPTIONAL extensions block. The rest of the query message is used for extensions to the original query format. The allowed extension types are GGEP, HUGE and XML (see Section 2.3 and Appendixes 1 and 2). If two or more of these extension types exist together, they are separated by a 0x1C (file separator) byte. Since GGEP blocks can contain 0x1C bytes, the GGEP block, if present, MUST be located after any HUGE and XML blocks. The type of each block can be determined by looking for the prefixes "urn:" for a HUGE block, "<" or "{" for XML and 0xC3 for GGEP. The extension block SHOULD NOT be followed by a null (0x00) byte, but some servents wrongly do that.

2.2.6 Query Hit

Query Hit messages has the following fields:

Bytes:	Description:
0	Number of Hits. The number of query hits in the result set (see below).
1-2	Port. The port number on which the responding host can accept incoming HTTP file requests. This is usually the same port as is used for Gnutella network traffic, but any port MAY be used.
3-6	IP Address. The IP address of the responding host. Note: This field is in big-endian format.
7-10	Speed The speed (in kb/second) of the responding host.
11-	Result Set. A set of responses to the corresponding Query. This set contains Number_of_Hits elements, each with the following structure:

Bytes:	Description:
0-3	File Index. A number, assigned by the responding host, which is used to uniquely identify the file matching the corresponding query.

```
        4-7     File Size. The size (in bytes) of the file whose
                index is File_Index.

        8-      File Name. The name of the file whose index is
                File_Index. Terminated by a null (i.e. 0x00)

        x       Extensions block. Allowed extension types are HUGE,
                GGEP and plain text metadata. This field is
                terminated by a null (0x00), even if there are no
                extensions (resulting in a double null). Also, the
                extensions block itself MUST NOT contain any null
                bytes.

                If two or more of these extension types exist
                together, they are separated by a 0x1C (file
                separator) byte. Since GGEP blocks can contain 0x1C
                bytes, the GGEP block, if present, MUST be located
                after any HUGE and plan text blocks.

                The type of each block can be determined by looking
                for the prefixes "urn:" for a HUGE block, 0xC3 for
                GGEP and anything else is probably plain text
                metadata.

                Plain text metadata is intended to be displayed
                directly to the user. It was first invented by
                Gnotella (a now discontinued Gnutella servent) to tag
                MP3 files. Examples:
                "192 kbps 44 kHz 3:23"
                "120 kbps(VBR) 44kHz 3:55" (variable bitrate)
                Other plan text formats MAY be used.

  x     RECOMMENDED extra block. This block is not required, but
        strongly recommended. It is sometimes called EQHD, or
        (incorrectly) just QHD. It has the following format:

        Bytes:
        0-3     Vendor Code. Four case-insensitive characters
                representing a vendor code. For example "LIME" for
                LimeWire. See registered codes and register yours at
                http://groups.yahoo.com/group/the_gdf/database?
                        method=reportRows&tbl=6
                        (Requires GDF membership)

        4       Open Data Size. Contains the length (in bytes) of the
                Open Data field. Set to 2 in most current
                implementations, and 4 in those that support XML
                metadata outside GGEP (see Section 2.3 and Appendix 2).
                The Open Data area MAY be larger to allow future
                extensions.

        x       Open Data. Contains two 1-byte flags fields with the
                following layout and in the specified order:

                bit:    Description:
                7,6     Reserved for future use
                5       flagGGEP
                4       flagUploadSpeed
                3       flagHaveUploaded
                2       flagBusy
                1       Reserved for future use
                0       flagPush

                The first flag byte can be viewed as an "enabler" for
                the flags in the second byte, the "setter".  Only
                those bits that were enabled must be considered by
                the servent as being valid.  This logic is reversed
                for flagPush, which is set in the first byte and
                enabled in the second.  The enabling byte allows
                you to know which flags are supported by a given
                servent.

                Bits 5,4,3,2 in the first byte MUST be set if and
                only if the corresponding flag in the second byte is
                meaningful.

                Bit 0 in the second byte MUST be set if and only
                if the corresponding flag in the second byte is
                meaningful. Yes, the order is reversed for this flag.
```

flagGGEP is set is set if and only if the private data block (see below) contains a GGEP block.

flagUploadSpeed is set if and only if the Speed field of the QueryHit message contains the highest average transfer rate (in kbps) of the last 10 uploads. Otherwise Speed field contains the hosts total upload speed as set by the user, and therefore less reliable.

flagHaveUploaded is set if and only if the servent has successfully uploaded at least one file.

flagBusy is set if and only if the all of the servent's upload slots are currently full.

flagPush is set if and only if the servent is firewalled or cannot accept incoming TCP connections for any other reason.

The reserved flags MUST not be set, unless they are used for a future extension.

If XML metadata (Appendix 2) is included in the current Query Hit message, the following 2 bytes of Open Data area will contain the size of the XML block. The XML block itself is placed in the private area (see below).

x Private Data. Undocumented vendor-specific data. This field continues till the servent Identifier, which uses the last 16 bytes of the message.

If the flagGGEP in the open data block is set, this block contains a GGEP (see Section 2.3) extension block. The GGEP block starts with a 0xC3 byte. Any data before or after the GGEP block is vendor-specific data, and MUST be ignored, if not recognized.

Servents are NOT RECOMMENDED to use the private data area for vendor specific data. Servents SHOULD use GGEP extensions instead.

If the Open Data area indicates an XML block is will also be placed in the private area (see Appendix 2). Assuming that the two bytes in the Open Data area specifies an XML block of m bytes, that block can be found by extracting the last m bytes of the private area. Both GGEP and XML can exist in the same Private Data area, but XML SHOULD be implemented inside GGEP.
[TODO: How about the nul after the XMP block? What is it good for?]

Last 16 Servent Identifier. A 16-byte string uniquely identifying the responding servent on the network. This SHOULD be constant for all Query Hit messages emitted by a servent and is typically some function of the servent's network address. The servent Identifier is mainly used for routing the Push Message (see below).

2.2.7 Use of Query and Query Hit

2.2.7.1 Forwarding and routing of Query and Query Hit messages

A servent SHOULD forward incoming Query messages to all of its directly connected servents, except the one that delivered the incoming Query. Servents using Flow control or Ultrapeers (sections 3.1 and 3.2) will not always forward every Query over every connection.

A servent MUST decrement a message header's TTL field, and increment its Hops field, before it forwards the message to any directly connected servent. If, after decrementing the header's TTL field, the TTL field is found to be zero, the message MUST NOT be forwarded along any connection.

A servent receiving a message with the same Payload Message and Message ID as one it has received before, MUST discard the message. It means the message has already been seen.

QueryHit messages MUST only be sent along the same path that carried the incoming Query message. This ensures that only those servents that routed the Query message will see the QueryHit message in response. A servent that receives a QueryHit message with Message ID = n, but has not seen a Query message with Message ID = n SHOULD remove the QueryHit message from the network.

2.2.7.2 When and how to send new Query messages.

Query messages are usually sent when the user initiates a search. A servent MAY also create Queries automatically, to find more locations of a resource for example. If doing so the servent MUST be very careful not overload the network. A servent SHOULD not send more than one automatic query per hour.

Servents SHOULD NOT allow the user to create a large amount of queries by repeatedly clicking on a button.

Servents SHOULD watch queries originating from its neighbours (Hops=0) If those queries are too frequent, are duplicates or indicate bad servents behavior in any other way, the servents SHOULD drop those queries or even close the connection.

The TTL value of a new query created by a servent SHOULD NOT be higher than 7, and MUST NOT be higher than 10. The hops value MUST be set to 0.

2.2.7.3 When and how to respond with Query Hit messages.

When a servent receives an incoming Query message it SHOULD match the Search Criteria of the query against its local shared files.

The Search Criteria is text, and it has never been specified which charset that text was encoded with. Therefore, servents MUST assume it is pure ASCII only. If any byte with the 7th bit set (high bit) is found, then either there is a GGEP extension specifying the encoding used, or the servent SHOULD guess the proper encoding. Most likely, it will be ISO-latin-1 or UTF-8.

Exactly how to interpret the Search Criteria is not specified either, but here are some guidelines for interoperability between servents:

The Search Criteria is a string of keywords. A servent SHOULD only respond with files that has all the keywords. It is RECOMMENDED to break up the words on any non-alphanumeric characters (anything but letters and numbers). A space is the standard separator between words.

Servents MAY also require that all matching terms be present in the same number and order as in the query.

Regular expressions are not supported and common regexp "meta-characters" such as "*" or "." will either stand for themselves or be ignored. The matching SHOULD be case insensitive. Empty queries or queries containing only 1-letter words SHOULD be ignored.

GGEP extensions MAY be used to provide details on how to parse the Search Criteria (such as specifying that regular expressions matching should be used), but a servent can never be sure other servents will understand the GGEP extension.

Servents MAY ignore queries whose Search Criteria is shorter than a chosen length. The reason is to ignore too broad searches.

Query messages with TTL=1, hops=0 and Search Criteria="   " (four spaces) are used to index all files a host is sharing. Servents SHOULD reply to such queries with all its shared files. Multiple Query Hit messages SHOULD be used if sharing many files. Allowed reasons not to respond to index queries include privacy and bandwidth.

Query Hit messages MUST have the same Message ID as the Query message it is sent in reply to. The TTL SHOULD be set to at least the hops value of the corresponding query plus 2, to allow the Query Hit to take a longer route back, if necessary. The TTL value MUST be at least the hops value of the corresponding query, and the initial hops value of the Query Hit message MUST (as usual) be set to 0. Some servents use a TTL of (2 * Query TTL + 2) in their replies to be sure that the reply will reach its destination. Replies with high TTL level SHOULD be allowed to pass through.

2.2.8 Push (0x40)

A Push message has the following fields:

Bytes:	Description:
0-15	Servent Identifier. The 16-byte string uniquely identifying the servent on the network who is being requested to push the file with index File_Index. The servent initiating the push request MUST set this field to the Servent_Identifier returned in the corresponding QueryHit message. This is used to route the Push message to the sender of the Query Hit message.
16-19	File Index. The index uniquely identifying the file to be pushed from the target servent. The servent initiating the push request MUST set this field to the value of one of the File_Index fields from the Result Set in the corresponding QueryHit message.
20-23	IP Address. The IP address of the host to which the file with File_Index should be pushed. This field is in big-endian format.
24-25	Port. The port number the receiver of this message should push to.
26-	OPTIONAL GGEP extension block. (see Section 2.3)

A servent may send a Push message if it receives a QueryHit message from a servent that doesn't support incoming connections. This might occur when the servent sending the QueryHit message is behind a firewall. When a servent receives a Push message, it SHOULD act upon the push request if and only if the servent_Identifier field contains the value of its servent identifier. The Message_Id field in the Message Header of the Push message SHOULD not contain the same value as that of the associated QueryHit message, but SHOULD contain a new value generated by the servent's Message_Id generation algorithm.

Push messages are forwarded back to the originator of the Query Hits message using the Servent Identifier value. This means multiple Push messages can have the same Servent Identifier. Push messages MUST only be considered as duplicates if the Message ID in the header is the same. Since Push messages are not broadcasted, duplicate messages should be very rare.

2.2.9 Bye (0x02)

The Bye message is an OPTIONAL message used to inform the servent you are connected to that you are closing the connection.

Servents supporting the Bye message MUST indicate that by sending the following header in the handshaking sequence:

```
        Bye-Packet: 0.1
```

Servents MUST NOT send Bye messages to hosts that has not indicated support using the above header. Future versions will be backwards compatible, so Bye messages MAY also be sent to hosts providing the above header with a later version number.

A Bye packet MUST be sent with TTL=1 (to avoid accidental propagation by an unaware servent), and hops=0 (of course).

A servent receiving a Bye message MUST close he connection immediately. The servent that sent the packet MUST wait a few seconds for the remote host to close the connection before closing it. Other data MUST NOT be sent after the Bye message. Make sure any send queues are cleared.

The servent that sent by Bye message MAY also call shutdown() with 'how' set to 1 after sending the Bye message, partially closing the connection. Doing a full close() immediately after sending the Bye messages would prevent the remote host from possibly seeing the Bye message.

After sending the Bye message, and during the "grace period" when we don't immediately close the connection, the servent MUST read all incoming messages, and drop them unless they are Query Hits or Push, which MAY still be forwarded (it would be nice to the network). The connection will be closed as soon as the servent gets an EOF condition when reading, or when the "grace period" expires.

A Bye message has the following fields:
Bytes: Description:

0-1 Code. The presence of the Code allows for automated processing of the message, and the regular SMTP classification of error code ranges should apply. Of particular interests are the 200..299, 400..499 and 500..599 ranges.
Here is the general classification ("User" here refers to the remote node that we are disconnecting from):

2xx The User did nothing wrong, but the servent chose to close the connection: it is either exiting normally (200), or the local manager of the servent requested an explicit close of the connection (201).

4xx The User did something wrong, as far as the servent is concerned. It can send packets deemed too big (400), too many duplicate messages (401), relay improper queries (402), relay messages deemed excessively long-lived [hop+TTL > max] (403), send too many unknown messages (404), the node reached its inactivity timeout (405), it failed to reply to a ping with TTL=1 (406), or it is not sharing enough (407).

5xx The servent noticed an error, but it is an "internal" one. It can be an I/O error or other bad error (500), a protocol desynchronization (501), the send queue became full (502).

2- NULL-terminated Description String. The format of the String is the following (refers to "\r" and to "\n"):

```
Error message, as descriptive as possible
```

or optionally, something more qualified, with HTTP-like headers giving out more information:

```
Error message, as descriptive as possible
Server: some server/version
X-Gnutella-XXX: some specific Gnutella header
for instance telling the host about alternate
nodes it could connect to
```

The presence of a at the end of message indicates that HTTP-like headers are present. The absence of any
 indicates that the short error message form was used.

Unless circumstances making that impossible (urgent disconnection due to a memory fault), the HTTP-like headers version SHOULD be used, with at least a Server: header, allowing better tracing and debugging.

For further information about the Bye message, please refer to the original documentation located at:
http://groups.yahoo.com/group/the_gdf/files/Proposals/BYE/

2.3 GGEP Extension blocks

The Gnutella Generic Extension Protocol (GGEP) allows arbitary extensions in Gnutella messages. A GGEP block is a framework for other extensions. If you wish to implement a new extension to a packet, you MUST do so inside GGEP. Some extensions that were invented before GGEP (XML metadata for example) are allowed to existoutside GGEP.

Servents are RECOMMENDED to implement GGEP. However, all servents MUST pass on GGEP extension blocks inside Gnutella messages. servents that have support the forwarding of all packets that contain GGEP extensions (whether or not they can process them), MUST include a new header in the Gnutella 0.6 connection handshake indicating this support. This will allow other servents to know what types of packets this servent can accept. The format of this header is

GGEP : majorversion.minorversion

As the current version of GGEP is 0.5 when this was written the header would be

GGEP: 0.5

Servents SHOULD remove any GGEP blocks from Ping, Pong and Push messages before sending those messages to hosts that have not indicated GGEP support.

For the original GGEP documentation see
http://groups.yahoo.com/group/the_gdf/files/Proposals/GGEP/

2.3.1 GGEP Format

A GGEP block always starts with a magic byte used to help distinguish GGEP extensions from legacy data which may exist. It must be set to the value 0xC3.

When a GGEP block is used between the nulls in a result in a Query Hits message, it is not allowed to contain any null bytes. This requires some special tricks in the field format.

The magic byte is followed by any number of extensions. They SHOULD be processed in the order in which they appear. The following is the format of each extension:

Bytes used:	Field Name:
1	Flags
1-15	ID
1-3	Data Length
x	Extension Data

```
Flags:           These are options which describe the encoding of the
                 extension header and data.

                 Bit    Name
                 7      Last Extension.  When set, this is the last
                        extension in the GGEP block.

                 6      Encoding.  The value contained in this field
                        dictates the type of encoding which should be
                        applied to the extension data (after possible
                        compression).

                        0 = There is no encoding on the data.
                        1 = The data is encoded using the COBS scheme.

                        Details about the COBS encoding scheme can be
                        found at http://www.acm.org/sigcomm/sigcomm97/
                                     papers/p062.pdf

                 5      Compression.  The value contained in this
                        field dictates the type of compression that
                        should be applied to the extension data.

                        0 = The extension data has not been compressed.
                        1 = The extension data should be decompressed
                            using the deflate algorithm.

                        One should only compress data if doing so will
                        make a material difference in the resulting
                        packet size.

                        Details about the Deflate compression scheme
                        may be found at http://www.gzip.org/zlib/
                        and http://www.faqs.org/rfcs/rfc1951.html

                 4      Reserved. This field is currently reserved for
                        future use.  It must be set to 0.

                 3-0    ID Len Value 1-15 can be stored here.  Since
                        this will not be zero, it ensures this byte
                        will not be 0x0.

ID:              The raw binary data in this field is the extension ID.
                 The length of this field can range between 1 and 15
```

```
                  bytes, and is determined by the Flags field. See
                  section 2.3.2 below on suggestions and rules for
                  creating extension IDs.  No byte in the extension
                  header may be 0x0.

Data Length:      This is the length of the raw extension data.  Please
                  note that most Gnutella clients will drop messages,
                  and possibly connections if the message size is
                  larger than a certain threshold (which varies
                  according to message type).  Please pay attention to
                  these limits when creating and bundling new
                  extensions.

                  This field uses an encoding technique that ensures
                  that 0x0 is never the value of any byte.  Steps were
                  also taken to ensure that the encoding is compact. In
                  this technique, a length field is the concatenation
                  of length chunks.  The format of each length chunk
                  (which contains 6 bits of length info) is described
                  in bit level below:

                  Format:
                  76543210
                  MLxxxxxx

                  M = 1 if there is another length chunk in the
                  sequence, else 0

                  L = 1 if this is the last length chunk in the
                  sequence, else 0

                  xxxxxx = 6 bits of data.

                  01aaaaaa ==> aaaaaa (2^6 values = 0-63)

                  10bbbbbb 01aaaaaa ==> bbbbbbaaaaaa
                  (2^12 values = 0-4095)

                  10cccccc 10bbbbbb 01aaaaaa ==> ccccccbbbbbbaaaaaa
                  (2^18 values = 0-262143)

                  Boundary Cases:
                  0      =                        01 000000b = 0x40
                  63     =                        01 111111b = 0x7f
                  64     =             10 000001  01 000000b = 0x8140
                  4095   =             10 111111  01 111111b = 0xbf7f
                  4096   = 10 000001  10 000000  01 000000b = 0x818040
                  262143 = 10 111111  10 111111  01 111111b = 0xbfbf7f

                  As you see, when the bits are concatenated, the
                  number is in big endian format.

Extension Data    The actual extension data.  The format of this field
                  varies between extensions.  A servent that does not
                  recognize and extension will not be able to parse the
                  Extension data, but since the length of this field is
                  specified by Data Length, it can still skip to the
                  next extension.  Note that extensions MAY be empty.
```

2.3.2 Creating Extension IDs

The Extension ID field in the GGEP header is a binary field consisting of between 1 and 15 bytes. It cannot contain the byte 0x0, and one must be able to compare IDs with a simple binary comparison. Aside from those rules, GGEP does not mandate any particular format, but does encourage the creation of short IDs that are free from conflicts. One should also note that Extension IDs are meant to be consumed by machines. Still, the following rules apply.

GDF Registered Extensions:
Any Extension ID of less than 4 bytes MUST be stored in the appropriate GDF database. Any Extension ID of less than 3 bytes must also be approved by the GDF. The format of the extension data must also be registered.

VendorID Extensions
This simple technique allows for the creation of ExtensionIDs based upon uses the following format VendorID.BinaryID

VendorID for a Gnutella servent is a 4 byte value that has been registered in the GDF Peer Codes database. In the QueryHit Descriptor, this case is case insensitive. With ExtensionIDs, the case matters, as one must be able to perform a binary comparison on the ID. This means an ExtensionID of "SWAP.1" and "swap.1" are different, but both "belong" the vendor who ones the code "SWAP."

This technique may be good for experimental and strictly vendor-specific extensions, but should be avoided for extension that may be useful for other vendors as well. Marking an extension by a vendor ID makes it harder for other vendors to use the extension in their servents.

Extension implementers SHOULD publish the ID, format, and expected data size for their extensions in the GDF database called "GGEP Extensions." located at
http://groups.yahoo.com/group/the_gdf/database?method=reportRows&tbl=10
(Requires GDF membership)

3 Protocol Usage

Apart from the protocol definition in section 2, there are also some guidelines on how to use the protocol. These are not absolutely necessary to participate in the network, but very important for an effective network.

3.1 Flow Control

It is very important that all servents have a system for regulating the data that passes through a connection.

The most simple way is to close a connection if it gets overloaded. A better way is to drop broadcasted packets to reduce the amount of bandwidth used. A much better way is to do the following:

Implement an output queue, listing pending outgoing messages in FIFO order. As long as the queue is less than, say, 25% of its max size (in bytes queued, not in amount of messages), do nothing. If the queue gets filled above 50%, enter flow-control mode. You stay in flow-control mode (FC mode for short) as long as the queue does not drop below 25%. This is called "hysteresis".

The queue size SHOULD be at least 150% of the maximum admissible message size.

In FC mode, all incoming queries on the connection are dropped. The rationale is that we would not want to queue back potentially large results for this connection since it has a throughput problem.

Messages to be sent to the node (i.e. queued on the output queue) are prioritized:

* For broadcasted messages, the more hops the packet has traveled, the less prioritary it is. Or the less hops, the more prioritary. This means your own queries are the most prioritary (hops = 0).

* For replies (query hits), the more hops the packet has traveled, the more prioritary it is. This is to maximize network usefulness. The packet was relayed by many hosts, so it should not be dropped or the bandwidth it used would become truly wasted.

* Individual messages are prioritized thusly, from the most prioritary to the least: Push, Query Hit, Pong, Query, Ping. The Bye message being special, it is always sent (i.e. the queue cannot be in FC mode since it needs to be cleared before sending Bye).

Normally, all messages are accepted. However, when the message to enqueue would make the queue fill to more than 100% of its maximum size, any queued message less prioritary in the queue is dropped. If enough room could be made, enqueue the packet. Otherwise, if the message is a Query, a Pong or a Ping, drop it. If not, send a Bye 502 (Send Queue Overflow) message.

Other known flow-control algorithms:

SACHRIFC is a simple, but still very effective flow control system that drops less important packets first. It can be found at:
http://groups.yahoo.com/group/the_gdf/message/5726

A more advanced flow control system can be found at:
http://www.grouter.net/gnutella/

3.2 Network Structure

[TODO: Ultrapeer is so important that the information required to implement it will be included here.]

Originally, all Gnutella nodes were connected to each other randomly. It worked fine for users with broadband connections, but not for users with slow modems. That problem can be solved by organizing the network in a more structured form.

3.2.1 Ultrapeer system

[TODO: Describe ultrapeer system here. Handshaking etc. Reference to QRP when used]
[TODO: Ultrapeer marked pongs: size field = power of 2]

The Ultrapeer system has been found effective for this purpose. It is a scheme to have a hierarchical Gnutella network by categorizing the nodes on the network as leaves and ultrapeers. A leaf keeps only a small number of connections open, and that is to ultrapeers. An ultrapeer acts as a proxy to the Gnutella network for the leaves connected to it. This has an effect of making the Gnutella network scale, by reducing the number of nodes on the network involved in message handling and routing, as well as reducing the actual traffic among them.

An ultrapeer only forwards a query to a leaf if it believes the leaf can answer it, and leaves never relay queries between ultrapeers. Ultrapeers are connected to each other and to "normal" Gnutella hosts (hosts that do not implement the Ultrapeer system).

An ultrapeer decides what queries to forward to leaf nodes using the Query Routing Protocol, QRP, which is described in section 3.2.2 below. If both an ultrapeer and a leaf node supports another protocol for deciding which queries are forwarded that MAY be used instead. QRP routing is not used between ultrapeers/normal hosts.

It is RECOMMENDED that servents implement the Ultrapeer system, or any future system for decreasing the bandwidth load on modem users.

For more information please read the original specification at:
http://www.limewire.com/developer/Ultrapeers.html

3.2.1.1 Ultrapeer election

Since Gnutella is a decentralized system, ultrapeers are elected without the use of a central server. It is up to each node to determine if it is to become an ultrapeer or a shielded leaf node. First, there are some basic requirements that must be satisfied to even consider becoming an ultrapeer.

* Not firewalled. This can usually be approximated by looking at whether the host has received incoming connections.

* Suitable operating system. Some operating systems handle large numbers of sockets better than others. Linux, Windows 2000/NT/XP, and Mac OS/X will typically make better ultrapeers than Windows 95/98/ME or Mac Classic.

* Sufficient bandwidth. At least 15KB/s downstream and 10KB/s upstream bandwidth is recommended. This can be approximated by looking at the maximum upload and download throughput.

* Sufficient uptime. Ultrapeers should have long expected uptimes. A reasonable heuristic is that the expected future uptime is proportional to the current uptime. That is, nodes should not become ultrapeers until they have been running at least a few hours.

* Sufficient RAM and CPU speed. Ultrapeers need memory for storing routing tables and CPU speed for outing all incoming queries. Exactly how much is needed depends how efficiently it is implemented and must be experimented with.

If the above criterias are met, a node is said to be ultrapeer capable. Note that this is not the same as actually being an ultrapeer.

Wheneither an ultrapeer capable node will actually become an
ultrapeer depends on if there is need for more ultrapeers on the
network, and on how well the above criterias are met. The need for
ultrapeers can be estimated from the noumber of ultrapeers found, and
can be communicated when new connections are established (see below).

3.2.1.2 Ultrapeer Handshaking

Ultrapeer capatibilities and information is exchanges during the
handskaking sequence when trying to establishing a new Gnutella
connection (see section 2.1). The following new headers are used:

* X-Ultrapeer: "True" signals that node is an ultrapeer, "False"
signals that the node wants to be a shielded leaf node.

* X-Ultrapeer-Needed: Used to balance the number of ultrapeers. [TODO: Write more about this one]

* X-Try-Ultrapeers: Like X-Try (see section 2.1), but contains only
addresses of ultrapeers.

* X-Query-Routing: Signals support for the Query Routing Protocol
(section 3.2.2). The header value is the QRP version (curretly 0.1).

It is important to note that headers can be sent in any order.
Also, case is ignored in "True" and "False".

Here is a sample interaction where a leaf connects to an ultrapeer.

```
Leaf                              Ultrapeer
-----------------------------------------------------------
GNUTELLA CONNECT/0.6
User-Agent: LimeWire/1.0
X-Ultrapeer: False
X-Query-Routing: 0.1

                                  GNUTELLA/0.6 200 OK
                                  User-Agent: LimeWire/1.0
                                  X-Ultrapeer: False
                                  X-Ultrapeer-Needed: False
                                  X-Query-Routing: 0.1
                                  X-Try: 24.37.144:6346,
                                   193.205.63.22:6346
                                  X-Try-Ultrapeers: 23.35.1.7:6346,
                                   18.207.63.25:6347

GNUTELLA/0.6 200 OK


[binary messages]                 [binary messages]
```

The leaf is now a shielded node of the ultrapeer. The leaf should
drop any non ultrapeer connections and send a QRP routing table
(assuming QRP is used).

If a shielded leaf node receives a connection request, it will refuse
to accept the connection by returning a 503 error code together with
X-Try and X-Try-Ultrapeer headers to redirect to remote host to other
addresses. For example, when a leaf tries to connect to another leaf
it may look like this. Non-essential headers have been removed in
this and the following examples.

```
Leaf1                             Leaf2
-----------------------------------------------------------
GNUTELLA CONNECT/0.6
X-Ultrapeer: False

                                  GNUTELLA/0.6 503 I am a leaf
                                  X-Ultrapeer: False
                                  X-Try: 24.37.144:6346
                                  X-Try-Ultrapeers: 23.35.1.7:6346


                                  [Terminates connection]
```

Sometimes nodes will be ultrapeer-incapable but unable to find an
ultrapeer. In this case, they behave exactly like old, unrouted
Gnutella 0.4 connections.

```
   Leaf1                             Leaf2
   -------------------------------------------------------------
   GNUTELLA CONNECT/0.6
   X-Ultrapeer: False

                                     GNUTELLA/0.6 200 OK
                                     X-Ultrapeer: False

   GNUTELLA/0.6 200 OK


   [binary messages]                 [binary messages]
```

When two ultrapeers meet, both set X-Ultrapeer: true. If both have
leaf nodes, they will remain ultrapeers after the interaction. Note
that no QRP route table is sent between ultrapeers after the
connection is established. Example handshake:

```
   UltrapeerA                        UltrapeerB
   -------------------------------------------------------------
   GNUTELLA CONNECT/0.6
   X-Ultrapeer: True

                                     GNUTELLA/0.6 200 OK
                                     X-Ultrapeer: True


   GNUTELLA/0.6 200 OK


   [binary messages]                 [binary messages]
```

Sometimes there will be too many ultrapeer-capable nodes on the
network. Consider the case of an ultrapeer A connecting to an
ultrapeer B. If B doesn’t have enough leaves, it may direct A to
become a leaf node. If A has no leaf connections, it stops fetching
new connections, drops any Gnutella 0.4 connections, and sends a QRP
table to B. Then B will shield A from all traffic. If A has leaf c
onnections, it ignores the guidance, as in the above case.

```
   UltrapeerA                        UltrapeerB
   -------------------------------------------------------------
   GNUTELLA CONNECT/0.6
   X-Ultrapeer: True

                                     GNUTELLA/0.6 200 OK
                                     X-Ultrapeer: True
                                     X-Ultrapeer-Needed: False


   GNUTELLA/0.6 200 OK
   X-Ultrapeer: False


   [binary messages]                 [binary messages]
```

3.2.2 Query Routing Protocol

The Query Routing Protocol (QRP for short) is an essential part of
the Ultrapeer specification: it governs how the Ultrapeer will filter
queries and only forward those to the leaf nodes most likely to have
a match. This is done without even knowing the resource names, by
looking the query words through a big hash table, that is sent by the
leaf node to its Ultrapeer.

The aim of the QRP is to avoid forwarding a query that cannot match,
it is not to forward only those queries that will match.

The overall operation goes thusly:

* At the leaf node level:

  + Break all the resource names into individual words. A word is
    made of a consecutive sequence of letters and digits.

  + Hash each word with a well-known hash function and insert a
    "present" flag in the corresponding hash table slot.
    Note that this hash table is a big array, and we don't store

the key, only the fact that a key ended up filling some slot. All words are lower-cased and all accents are removed from them, i.e. "déjà" is transformed into "deja", so that only ASCII characters remain. Only those words that are made of at least 3 letters are retained.

+ All words are re-hashed with their trailing 1, 2, or 3 letters removed, provided the word length after such trimming is at least 3 letter long. This is a simple attempt to remove plural from words. Optionally, nodes can chop off more letters from the end, provided that each hashed word is at least 3 character long.

+ The "boolean vector" built at later stage is optionally compressed, broken up in small messages, and sent mixed with regular Gnet traffic to the ultrapeer.

* At the Ultrapeer level:

  + Until the whole "boolean vector" is received from a leaf node, all queries are forwarded to that node.

  + When the "boolean vector" is fully received, it is going to be used as the Query Routing table for that leaf node: queries are broken into individual words, all accentuated letters are removed.

  + For each leaf node with a Query Routing table:

    . Each word is then hashed and looked up in the Query Routing table.

    . Depending on the query matching rules (see 2.2.7.3), either ALL the words will be required to be found in the Query Routing, or only some of them, to declare a Query Routing Hit.

    . Only those queries that were declared a Hit at the previous stage will be forwarded to a given leaf node.

The remaining sections define the hashing function, the mechanism used to build up the "boolean vector" and compress it, the protocol to transmit the vector to the Ultrapeer, and finally give operating hints for the table sizing.

[TODO: finish the QRP description --RAM]

The Query Routing Protocol (QRP) used in Ultrapeer can be found at:
http://www.limewire.com/developer/query_routing/keyword%20routing.htm

4 File Transfer

4.1 Normal File Transfer

Once a servent receives a QueryHit message, it may initiate the direct download of one of the files described by the message's Result Set. Files are downloaded out-of-network i.e. a direct connection between the source and target servent is established in order to perform the data transfer. File data is never transferred over the Gnutella network.

The file download protocol is HTTP. It is RECOMMENDED to use HTTP 1.1 (RFC 2616), but HTTP 1.0 (RFC 1945) can be used instead. The full specifications are available in those RFCs. The following includes only the basic things. The following examples assumes that HTTP 1.1 is used.

The servent initiating the download sends a request string on the following form to the target server:

```
GET /get// HTTP/1.1
User-Agent: Gnutella
Host: 123.123.123.123:6346
Connection: Keep-Alive
Range: bytes=0-
```

where and are one of the File Index/File Name pairs from a QueryHit message's Result Set. For example, if the Result Set from a QueryHit message contained the entry

```
File Index: 2468
```

```
    File Size: 4356789
    File Name: Foobar.mp3
```

then a download request for the file described by this entry would be initiated as follows:

```
    GET /get/2468/Foobar.mp3 HTTP/1.1
    User-Agent: Gnutella
    Host: 123.123.123.123:6346
    Connection: Keep-Alive
    Range: bytes=0-
```

Servents MUST encode the filename in GET requests according the standard URL/URI encoding rules. Servents MUST accept URL-encoded GET requests. Since some old servents does not support encoding, servents SHOULD accept non-encoded requests and MAY try a non-encoded requests if a 404 Not Found error is returned for the initial request.

The Host header is required by HTTP 1.1 and specifies what address you have connected to. It is usually not used by the receiving servent, but its presence is required by the protocol.

The allowable values of the User-Agent string are defined by the HTTP standard. Servent developers cannot make any assumptions about the value here. The use of 'Gnutella' is for illustration purposes only.

The server receiving this download request responds with HTTP 1.1 compliant headers such as

```
    HTTP/1.1 200 OK
    Server: Gnutella
    Content-type: application/binary
    Content-length: 4356789
```

The file data then follows and should be read up to, and including, the number of bytes specified in the Content-length provided in the server's HTTP response.

Note: Servents SHOULD use HTTP version 1.1 for file transfer, but some support only HTTP version 1.0. Servents MUST accept incoming HTTP/1.0 requests, and SHOULD retry with HTTP/1.0 if the remote host is not HTTP/1.1 compliant.

Though it is strongly RECOMMENDED to have full HTTP/1.1 support, some servents do not. The most important features for Gnutella, range requests and Persistent Connections MUST be supported. Some old servents, however, do not.

Range requests are on the form

```
    GET /get/2468/Foobar.mp3 HTTP/1.1
    User-Agent: Gnutella
    Host: 123.123.123.123:6346
    Connection: Keep-Alive
    Range: bytes=4932766-5066083
```

Note that the Range header does not have to specify both start and end positions. The response is on the form

```
    HTTP/1.1 206 Partial Content
    Server: Gnutella
    Content-Type: audio/mpeg
    Content-Length: 133318
    Content-Range: bytes 4932766-5066083/5332732
```

The Connection header tells the remote host if the connection should be closed when the transfer is finished or not. "Connection: close" means that the connection MUST be closed after the transfer. "Connection: Keep-Alive" or no Connection header means the connection MUST be kept open. The client MAY then issue another request for another range or another file. The request MAY be sent before the previous transfer is finished. Persistent Connections is described in section 8.1 of RFC 2616.

Headers unknown to the servent MUST be quietly ignored.

Servents SHOULD NOT attempt to download multiple files from the same source at once. Files SHOULD be locally queued instead.

Servents are also RECOMENDED to use and understand the http extension described in HUGE. (see Appendix 1)

4.2 Firewalled servents

It is not always possible to establish a direct connection to a Gnutella servent in an attempt to initiate a file download. The servent may, for example, be behind a firewall that does not permit incoming connections to its Gnutella port. If a direct connection cannot be established, the servent attempting the file download may request that the servent sharing the file "push" the file instead. A servent can request a file push by routing a Push request back to the servent that sent the QueryHit message describing the target file. The servent that is the target of the Push request (identified by the Servent Identifier field of the Push message) SHOULD, upon receipt of the Push message, attempt to establish a new TCP/IP connection to the requesting servent (identified by the IP Address and Port fields of the Push message). If this direct connection cannot be established, then it is likely that the servent that issued the Push request is itself behind a firewall. In this case, file transfer cannot take place by the means of what is described in this document.

If a direct connection can be established from the firewalled servent to the servent that initiated the Push request, the firewalled servent should immediately send the following:

```
    GIV :/
```

Where and are the values of the File Index and Servent Identifier fields respectively from the Push request received, and is the name of the file in the local file table whose file index number is . The File Name MAY be url/uri encoded. The servent that receives the GIV (the servent that wants to receive a file) SHOULD ignore the File Index and File Name, and request the file it wants to download. The servent that sent the GIV MUST allow the client to request any file, and not just the one specified in the Push message. The GET request and the remainder of the file download process is identical to that described in the section 4.1 (Normal File Transfer) above.

The is formatted as hexadecimal, and must be read case-insensitively. For instance:

```
    GIV 36:809BC12168A1852CFF5D7A785833F600/Foo.txt
    GIV 124:d51dff817f895598ff0065537c09d503/Bar.html
```

If the TCP connection is lost during a Push initiated file transfer, it is strongly RECOMMENDED that the servent who initiated the TCP connection (the servent providing the file) attempt to re-connect. That is important, since the servent receiving the file might not be able to get another Push message to the servent providing the file.

4.3 Busy Servents

Servents whose upload bandwidth is already saturated with transfers MAY reject a download request by returning the 503 response code. Servents MAY simply have a fixed number of available upload slots, but SHOULD use a system that utilizes upload bandwidth better. Allowing new downloads as long as 20% of total upload bandwidth is unused is one possibility.

Busy servents receiving a Push message SHOULD connect to the host requesting a push, and return the 503 Busy code when the remote host has requested the file.

Servents MAY try requesting a download again when the servent providing the file returns the busy code, but MUST not do so more often than once per minute and file source. That means a Servent MUST NOT open new connections to a remote host more than once per minute. Servents SHOULD prevent other servents breaking the above rule from increasing their chanses to downlaoad a file. This can for example be archived by refusing any connection attempts from a particular host if a download request has been denies less than 50 seconds ago, or by adding hosts that request too often to a ban list.

Servents MAY use queuing systems to allow downloaders to stand in queue to download a file, but that is outside the scope of this document.

If a transfer is interrupted, the serving servent SHOULD keep the allocated slot/bandwidth reserved for at least one minute. The downloader would then be allowed to reconnect and resume the transfer.

4.3.1 Upload queueing

[TODO: Write about Shareaza style upload queues (optional)]
[TODO: Rewrite this (copied from GDF post)]

Clients which support queues send "X-Queue: 0.1", which simply tags the request as a candidate for queuing. If this header is not received, the requesting client is assumed to follow normal Gnutella behavior in the event of a busy response.

If there is an upload "slot" available, the download begins as normal with a 200 or 206 response. If not, the request is placed at the end of the queue and a 503 response is returned with the additional X-Queue header, of the form:

```
X-Queue: position=2,length=5,limit=4,pollMin=45,pollMax=120
```

Clearly this header includes several pieces of information separated by commas in the usual manner. Every part is optional, and if desired it can be broken into multiple headers, etc. Anyway, the parts:

The "position" key indicates the request's position in the queue, where position 1 is next in line for an available slot.
The "length" key indicates the current length of the queue, for informational purposes. Likewise the "limit" key specifies the number of concurrent uploads allowed. All of this information is completely optional, and is only used for display within the client.

Finally, "pollMin" and "pollMax" provide hints to the requesting client as to how often it should re-request the file (in seconds). Requesting more often than pollMin will be seen as flooding, and cause a disconnection. Failing to issue a request before pollMax will be seen as a dropped connection. Once again these items are optional and need not be present in the header, in which case a default retry interval can be used.

Upon receiving a 503 response with an X-Queue header, the downloader displays any information it received to the user and waits for an appropriate period before reissuing the request. The default retry period is adjusted to lie comfortably within pollMin and pollMax if they were present in the response, which allows a particularly busy server to adjust its parameters and reduce load. When the request finally succeeds, it does so in the normal way.

[TODO: End of rewrite this]

4.4 Sharing

Servents that are able to download files MUST also be able to share files with others. Servents SHOULD encourage users to share files.

Servents SHOULD attempt to prevent programs that are not able to share files from downloading files. This means that servent SHOULD not allow uploads to web browsers and download accelerators. The User-Agent http header tells what program the remote host is running. Many servents return a html page instead, telling the user how Gnutella works, and where to get a servent.

Servents MUST NOT give precedence to other users using the same servent. They MUST answer Query messages and accept file download requests using the same rules for all servents. Servents MAY, however, attempt to block servents that do not follow the rules in this protocol in way that seriously hurts others experience of the Gnutella network.

Servents SHOULD, by default, share the directory where downloaded files are placed. Servents SHOULD also share new downloaded files without waiting for the servent to be restarted. Servents SHOULD avoid changing the index numbers of shared files.

Servents MUST NOT share partially downloaded (incomplete) files as if

they were complete. This is often done by using a separate directory
for incomplete downloads. When the download finishes, the file is
move to the downloads directory (that should be shared). Partial
files MAY be shared in a way the makes it clear to other servents
that the file is incomplete.

5 Security Considerations

[TODO: This section is very incomplete. Any suggestions are welcome.]

5.1 Threats against individual Gnutella participants

[TODO: Write about threats against individual Gnutella participants Such as flooding,
fake files, DoS, etc. Flooding hostcache with faked pongs]
[TODO: How one can protect oneself and other gnet users]

Inexperienced users might share sensitive information, such as cookie
or password files, on the Gnutella network. Servents SHOULD warn
users who try share such information.

Malicious file, such as viruses and trojans, might be shared on the
Gnutella network by malicious or unexpecting users. Servents SHOULD
encourage users to scan downloaded files for viruses etc. but this is
outside the scope of the Gnutella protocol.

5.2 Threats against the Gnutella network

[TODO: Write about threats against the Gnutella network
Such as query flooding, fake files, DoS, fake random pongs etc.]
[TODO: What a servent should do to protect the network]

[TODO: A solution is fair sharing of bandwidth between connections and message types]

5.3 Threats against third parties

[TODO: Write about threats against third parties. Such (D)DoS, etc.]
[TODO: How to avoid]

Would it be possible to use the power of the Gnutella network to
attack internet hosts? That issue will be discussed in this section.

The ways of doing so an attacker might try is:

* Responding to Ping messages with Pong messages containing the
  IP address and port of the target host. This would cause other
  Gnutella servents to attempt to connect to the target host,
  thinking it is a Gnutella host.
  [TODO: Can this really be an effective attack? Would the target receive that many
connection attempts?]

* Responding to Query messages with Query Hit messages containing
  the IP address and port of the target host.
  [TODO: This might actually be an issue. How about recommending servents to attempt to drop
faked QHs somehow?]

* Responding to Query Hit messages with Push messages containing
  the IP address and port of the target host. This would cause
  the servent receiving the Push message to attempt a connection
  to the target host. It would, however, not be more than one
  connection per Push message, so it could not be used to launch
  large Denial-of-Service attacks.
  [TODO: Is that correct, or is it a real threat?]

6 Credits

The authors would like to thank:
    [TODO fill]

New features mentioned in this document, not present in the original
0.4 Gnutella specification document, published by the now defunct
Clip2 company should be credited thusly:

0.6 Handshaking Protocol	LimeWire LLC
X-Try Header	Mike Green
Bye Packet	Raphael Manfredi
Pong Caching	LimeWire LLC and others
EQHD Block	Free Peers Inc.

GGEP	Jason Thomas
HUGE	Gordon Mohr
Ultrapeers	LimeWire LLC
Query Routing Protocol	LimeWire LLC
XML queries	LimeWire LLC
Negotiation of Gnet Compression	Raphael Manfredi

[TODO missing?]

Appendix 1: HUGE (Hash/URN Gnutella Extensions)

HUGE is used to provide capability for hashes (numbers uniquely identifying files) and other urn:s to Gnutella. HUGE SHOULD be implemented inside GGEP (Section 2.3), but can also be used as a stand-alone extension block. When inside GGEP, the GGEP extension-identifier for HUGE info is 'u'. When used as a stand-alone extension, HUGE blocks start with "urn:". There MAY be multiple HUGE blocks in one Gnutella message (separated by 0x1C bytes).

HUGE also extends the direct file transfer between hosts, to allow communication or urn:s, and to build "download meshes" that inform servents of other locations of a file.

The HUGE documentation is available at:
http://groups.yahoo.com/group/the_gdf/files/Proposals/HUGE

Servents are RECOMMENDED to implement HUGE.

Appendix 2: XML

XML blocks can be used to issue rich queries and to include metadata about files in query hit messages. XML SHOULD be implemented inside GGEP, but can also be used as a stand-alone extension block.

If there is a "{deflate}" before the XML block it is compressed using the defalte algorithm. "{plaintext}" or no such prefix means uncompressed plaintext XML. Any other prefix mean the XML block is compressed using an algorithm that is not known at this time.

XML blocks can be recognized by them starting with "<" or "{".
(Do not rely on "

Note that although we only specify "deflate" here, the servent MAY advertise the set of various compression algorithms it knows, subsequent items being separated by a ",".

And to accept compression, the other side acknowledges by sending:

```
   Content-Encoding: deflate
```

The servent just picks the compression scheme it supports amongst the ones advertised by the remote end in the Accept-Encoding line. The Content-Encoding MUST contain only one value.

This also means that compression settings is asymmetric: a node can send compressed data but receive uncompressed data.

Here's an example where both nodes support compression, comments starting with "--", and ending removed for clarity:

```
   GNUTELLA CONNECT/0.6
   Accept-Encoding: deflate       -- OK for reception of compressed data

       GNUTELLA/0.6 200 OK
       Accept-Encoding: deflate   -- I can also receive compressed data
       Content-Encoding: deflate  -- And I will send compressed data

   GNUTELLA/0.6 200 OK
   Content-Encoding: deflate      -- OK, will also compress data
```

Here's an example where compression will only be made on the transmission side of the first node (A is the node initiating the handshake, B is the node replying):

```
   GNUTELLA CONNECT/0.6
   Accept-Encoding: deflate      -- OK for reception of compressed data
```

```
        GNUTELLA/0.6 200 OK
        Accept-Encoding: deflate  -- I can also receive compressed data
                                   -- I refuse to compress data, sorry

    GNUTELLA/0.6 200 OK
    Content-Encoding: deflate     -- OK, I will compress data sent
                                   -- But I will receive uncompressed data

    B is compressed, flow from B->A is not>

 Even though GGEP payloads (see Section 2.3) can be compressed, and
 this information is visible in the GGEP header, it is not advisable
 to decompress those payloads before sending them to the compressing
 layer.  The deflate algorithm does not expand already-compressed data
 by a large factor and emits them as clearly marked non-compressible
 data (the overhead is limited to roughly 0.1%). If connection
 compression is widely used on the Gnutella network, individual GGEP
 extensions SHOULD NOT be compressed.



 [TODO: Some spelling errors.]
 [TODO: Many of the docs referred to are still drafts]

 [TODO: RFCs have 72 char lines, and a 3 char left margin for most text blocks.
 I intend to make lines max 69 chars, and text blocks will be indented 3 chars.]
 [TODO: Break up into pages (like RFCs), but not before the final release (Or convert to XML)]

 [HUGE should perhaps be integrated]
 [The XML spec should not be integrated. We just specify where there is XML. The XML format that
 limewire uses is not a part of Gnutella. Anyone can use any XML format.]
```