Gnutella Protocol Development

The Annotated Gnutella Protocol Specification v0.4 (1)

Document Revision 1.6
Status: Annotated Standard
Based on previous Revision 1.2

Clip2
The Gnutella Developer Forum (GDF)
http://groups.yahoo.com/group/the_gdf

Summary

Gnutella (2) is a protocol for **distributed search**. Although the Gnutella protocol supports a traditional client/centralized server search paradigm, Gnutella's distinction is its *peer-to-peer*, decentralized model. In this model, every client is a server, and vice versa. These so-called Gnutella **servents** perform tasks normally associated with both clients and servers. They provide client-side interfaces through which users can issue queries and view search results, while at the same time they also accept queries from other servents, check for matches against their local data set, and respond with applicable results. Due to its distributed nature, a network of servents that implements the Gnutella protocol is highly fault-tolerant, as operation of the network will not be interrupted if a subset of servents goes offline.

---

(1)

This document represents the *de facto* standard Gnutella 0.4 protocol. However, several implementations have extended the descriptors that comprise the protocol, and have imposed additional rules on the transmission of these descriptors through the Gnutella network. Known extensions to the protocol are provided in an Appendix at the end of this document, but some variations not documented here may be encountered in practice.

(2)

Typically pronounced "new-tella" or, less commonly, "guh-new-tella".

Table of contents

---

1. Protocol Definition

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

Descriptor	Description
Ping	Used to actively discover hosts on the network. A servent receiving a Ping descriptor is expected to respond with one or more Pong descriptors.

Pong	The response to a Ping. Includes the address of a connected Gnutella 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 descriptor will respond with a QueryHit if a match is found against its local data set.
QueryHits	The response to a Query. This descriptor 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.

A Gnutella servent connects itself to the network by establishing a connection with another servent currently on the network. The acquisition of another servent's address is not part of the protocol definition and will not be described here (Host cache services are currently the predominant way of automating the acquisition of Gnutella servent addresses).

---

2. Connection Procedure and Protocol Negociation

Once the address of another servent on the network is obtained *(its IPv4 address and its port number)*, a TCP connection to the servent is created, and the following Gnutella connection request string (ASCII encoded) may be sent:

```
GNUTELLA CONNECT/<protocol version string>\n\n
```

where <protocol version string> is defined to be the ASCII string "0.4" (or, equivalently, "\x30\x2e\x34") in **this version** of the specification, and where "\n" represent an ASCII line-feed character (LF, code 0xa=10).

Note:

Compliant servents MAY accept a CR+LF termination instead of LF+LF, but MUST not generate a CR+LF terminator when using this 0.4 version of the Gnutella connection protocol.
With the Gnutella protocol, the "GNUTELLA" initial verb is mandatory to avoid collision with standard or optional HTTP and FTP verbs, and the final token "CONNECT/<protocol version string>" qualifies BOTH the connection protocol for Gnutella, AND the implicit set of standard descriptors and their semantics defined by the current protocol specification.
The above string unambiguously defines the first interoperable version of the Gnutella connection protocol. It MUST be accepted by all Gnutella compatible servents, even if the current version 0.4 of the Gnutella connection protocol has been obsoleted by the version 0.6 which introduces optional mechanisms comparable to HTTP/1.1.
Deprecated legacy applications using the Gnutella may use another connection string for private use, but they are not interoperable (if authentication is required to build a private network, implementors SHOULD use the newer 0.6 version of the Gnutella connection protocol to implement it with connection headers).

A servent wishing to accept the connection request MUST respond with:

```
GNUTELLA OK\n\n
```

Any other response indicates the servent's unwillingness to accept the connection. A servent may reject an incoming connection request for a variety of reasons - a servent's pool of incoming connection slots may be exhausted, or it may not support the same version of the protocol as the requesting servent, for example.

If a connection request must be rejected by a Gnutella compliant servent, it MAY use an HTTP-like status line (starting with a numeric status code such as "404 Not found").

Note:

Compliant servents MAY accept incoming CR+LF termination instead of LF+LF, but MUST not generate a CR+LF terminator when using this 0.4 version of the Gnutella connection protocol, as the result is unpredictable, and the protocol would non synchronize properly.
The initial "GNUTELLA" verb before the response avoids collision with HTTP servers.
Some servent implementations respond with "GNUTELLA CONNECT/0.4\n\n", anticipating a protocol negociation. However, as there was no common standard before the 0.4 protocol, such specification was not necessary. Servents MAY accept it only for legacy support with previous versions. This 0.4 version of the protocol does not support protocol version negociation, which has been introduced in the later 0.6 version of the Gnutella connection protocol defined in another document.

---

3. Peer-to-Peer Gnutella Packets: Descriptors

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

Note 1:

All fields in the following structures are in **little-endian** byte order unless otherwise specified.
This is differing from the traditional network-byte-order traditionally used in other networking protocols, but this has been kept for historical reasons and interoperability with existing Gnutella servents.
The traditional byte ordering ntoh(), ntol(), hton(), lton() functions used in networking libraries MUST NOT be used for descriptors as they assume a big-endian byte order for the network encoding (these functions or macros are no-operation identity only on big-endian machines, such as Motorola systems, and perform byte swaps on Intel systems). These functions MUST be replaced by providing functions like nltoh(), nltol(), htonl(), ltonl() assuming little-endianness on the network.

Note 2:

All 32-bit IP addresses in the following structures are in IPv4 format. For example, the IPv4 byte array:

Byte value	0xD0	0x11	0x32	0x04
Byte offset	0	1	2	3

represents the dotted address IPv4 "208.17.50.4". I.e. network addresses use the standard network byte-order, defined as **big-endian** for IPv4.

3.1. Descriptor Header

Fields	Descriptor ID	Payload Descriptor	TTL	Hops	Payload Length
Byte offset	0...15	16	17	18	19...22

Descriptor ID

A 16-byte string **uniquely** identifying the **descriptor** on the network. Its value must be preserved when forwarding messages between servents. Its use allows detection of cycles and help reduce unnecessary traffic on the network.

When generating 128-bit Descriptor IDs, servents can use the UUID generation algorithm, or use a cryptographically strong random generator. The value of the Descriptor ID carries no signification, and should always be treated as an opaque binary string, whose byte order must be preserved when forwarding messages.

However, within Pong descriptors, which uniquely identifies a servent host identified by a stable GUID in a more reliable and persistent way than by its current IP and port number address, a special Pong marking is required for newer applications: byte 8 SHOULD be set to 0xFF (indicating that the GUID unambiguously and uniquely identifies the servent), and byte 15 SHOULD be set to 0x00 for future use.

Payload Descriptor

0x00 = Ping (see section 3.2.1)
0x01 = Pong (see section 3.2.2)
0x80 = Query (see section 3.2.3)
0x81 = QueryHits (see section 3.2.4)
0x40 = Push (see section 3.2.5)

Extension Descriptors:

0x02 = Bye (see section 3.2.6)
0x10 = IBCM (Reserved for the non-standard InBandControlMessage descriptor, but MAY cause compatibility problem with legacy, non IBCM-aware, servents)
0x30 = QRP (see section 3.2.7)
0x31 = Open Vendor Extension (see section 3.2.8)
0x32 = Standard Vendor Extension (see section 3.2.9)

Note: Other values SHOULD not be used for now, as remote servents may consider it as invalid. Their use will be specified in an higher version of the protocol than the current 0.4 protocol (or its 0.6 extension).

TTL

Time To Live. The number of times the descriptor will be forwarded by Gnutella servents before it is removed from the network.

Each servent MUST decrement the *TTL* before passing it on to another servent. When the *TTL* reaches 0, the descriptor MUST no longer be forwarded.

Note: This field is unsigned, however a value higher than 127 will very probably be considered as excessive when used on the Internet.

Hops

The number of times the descriptor has been forwarded.

Note: This field is unsigned, however a value higher than 127 will very

probably be considered as excessive when used on the Internet.

As a descriptor is passed from servent to servent, the *TTL* and *Hops* fields of the header MUST satisfy the following conditions:

$$TTL_{(i)} + Hops_{(i)} = TTL_{(0)}$$
$$\mathrm{TTL}_{(i+1)} < \mathrm{TTL}_{(i)}$$
$$\mathrm{Hops}_{(i+1)} > \mathrm{Hops}_{(i)}$$

where $TTL_{(i)}$ and $Hops_{(i)}$ are the value of the TTL and Hops fields of the header at the descriptor's i^{-th} hop, for $i >= 0$.

Payload Length

The length of the descriptor immediately following this header. The next descriptor header is located exactly *Payload Length* bytes from the end of this header i.e. there are no gaps or pad bytes in the Gnutella data stream.
Note: This field is unsigned however a value of 2GB or more will very probably be considered as excessive. With the current specification of the protocol, the last encoded byte of the Payload Length field SHOULD then be 0 (as Payloads won't reach 16MB when used on the Internet).

The *TTL* is the only mechanism for expiring descriptors on the network. Servents SHOULD carefully scrutinize the *TTL* field of received descriptors and lower them as necessary. Abuse of the *TTL* field will lead to an unnecessary amount of network traffic and poor network performance.

Note:

Some servents MAY consider excessive values for TTL+Hops as indicating desynchronization of the connection input stream. Also, a descriptor where TTL=0 and Hops=0 is invalid. All servents MUST consider that TTL+Hops values between 1 and 7 are valid (a higher range is possible but not recommended for use on the Internet). A servent MAY reduce excessive TTL value, but MUST NOT increase it when forwarding or caching Descriptors. A servent MUST NOT reduce the Hops value as this will break the discovery of shorter routes and will affect route caches. When forwarding a descriptor to remote servents connected with slow or unreliable connections, a servent MAY also count more than 1 Hop and reduce the TTL by the equivalent number, provided that the resulting TTL value does not reach 0 (in such a case the descriptor MUST be discarded).

The *Payload Length* field is the ONLY reliable way for a servent to find the beginning of the next descriptor in the input stream. The Gnutella protocol does NOT provide an "eye-catcher" string or any other descriptor synchronization method *(it assumes that reliable TCP connections are used)*. Therefore, servents SHOULD **rigorously** validate the *Payload Length* field for each descriptor received (at least for fixed-length descriptors). If a servent becomes out of synch with its input stream, it SHOULD drop the connection associated with the stream since the upstream servent is either generating, or forwarding, invalid descriptors.

Note:

A desynchronization MAY be detected by the presence of an unknown value for the Payload Descriptor field in a single descriptor message, which servents are NOT required to silently discard.
For example, a new Payload Descriptor value has been proposed, the "Bye" descriptor with value 0x02, which gives the reason why a servent is being disconnected. The currently defined policy with unknown Payload Descriptors allows this because this message will not be followed by any other Descriptor, so the connection MAY still be silently dropped. This is however a proposed extension, whose payload format has still not been agreed upon among servents implementors. Its specification is not part of this document, and MAY be documented later.

3.2. Descriptor Payloads

Immediately following the descriptor header, is an optional payload, whose content and structure depends on the *Descriptor Payload* field in the descriptor header. The following sections detail them:

3.2.1. Ping (0x00) Descriptor Payload

Fields	*Optional Ping Data*
Byte offset	*0...L-1*

Optional Ping Data

This is an optional field consisting in bytes of variable length, it is reserved for extensions of the current version of the protocol, to specify filters about expected Pong replies. Its maximum length is bounded by the Payload Length field of the header.
When used, this field SHOULD be small and agreed upon with other Gnutella servent implementors, as this field MAY be specified in a further specification of the protocol.

Standard *Ping* descriptors currently have no associated payload and are of zero length. A *Ping* is simply represented by a descriptor header whose *Payload Descriptor* field is 0x00 and whose *Payload Length* field is 0x00000000.

A servent uses *Ping* descriptors to actively probe the network for other servents. A servent receiving a *Ping* descriptor MAY elect to respond with a *Pong* descriptor, which contains the address of an active Gnutella servent (possibly the one sending the *Pong* descriptor) and the amount of data it’s sharing on the network.

This specification makes no recommendations as to the frequency at which a servent SHOULD send *Ping* descriptors, although servent implementers SHOULD make every attempt to minimize *Ping* traffic on the network.

Note:

There's no requirement to always forward any Ping request

to other connected servents or with a large TTL+Hops value. So, most actual servents implement a traffic limiting policy for Ping descriptors.

3.2.2. Pong (0x01) Descriptor Payload

Fields	Port	IP Address	Number of Files Shared	Number of Kilobytes Shared	*Optional Pong Data*
Byte offset	0...1	2...5	6...9	10...13	*14...L-1*

Port

The TCP port number on which the responding host can accept incoming Gnutella connections. (See section 3.2.2.2 below)

IP Address

The IPv4 address of the responding host. (See section 3.2.2.3 below)
This field is in **big-endian** format.

Number of Files Shared

The number of files that the servent with the given *IP Address* and *Port* is sharing on the network.
Note: An excessive number of shared files will sometimes be ignored by servents receiving it, because it is suspect or because cumulating it could produce internal overflows.
*This **informative** field can be null but some servents have local policies that restrict accesses from "freeloaders" that don't share a minimum number of files.*

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.
Note: An excessive total shared size (more than 2GB), or an excessive mean size per shared file, will sometimes be ignored by servents receiving it because it is suspect or because cumulating it could produce internal overflows.
*This **informative** field can be null but some servents have local policies that restrict accesses from "freeloaders" that don't share a minimum volume of files.*

Optional Pong Data

This is an optional field of variable length, it is reserved for extensions of the current version of the protocol, to give other information about the servent, or to provide alternate transport protocols or addresses that allow incoming connections to the servent. Its maximum length is bounded by the Payload Length field of the header.
When used, this field SHOULD be small and agreed upon with other Gnutella servent implementors, as this field MAY be specified in a further specification of the protocol.

Pong descriptors are ONLY sent in response to an incoming *Ping*

descriptor. Multiple *Pong* descriptors MAY be sent in response to a single *Ping* descriptor. This enables host caches to send cached servent address information in response to a *Ping* request.

3.2.2.1. Usage policy

1) **Fields that SHOULD be preserved from incoming Pongs:**
In order to reduce the network traffic used by *Pong* descriptors and to discover shorter or alternate routes to the same servent, the *Descriptor ID* field of cached *Pongs* SHOULD be preserved locally along with the Hops and TTL fields.
However, excessive *Hops+TTL* values in incoming SHOULD be reduced by keeping the *Hops* field. If it has the effect of producing a negative or null TTL value, the Pong MAY be marked as invalid and be discarded, as the corresponding advertized servent may be unreachable via the Gnutella network.

2) **Generating the Descriptor Id for Pong descriptors:**
The *Descriptor Id* associated to the payload information of *Pong* replies SHOULD be constant for all *Ping* requests received from the same or alternate connection, at least as long as the responding servent has an active connection to the network, unless the servent implements multiple listening IP interfaces attached to distinct networks, considered as if it was different servents.
This *Descriptor Id* SHOULD be globally unique for that server instance. So its generation should use the UUID algorithm or a cryptographically strong random generator. However byte 8 SHOULD be set to 0xFF (indicating that the GUID unambiguously and uniquely identifies the servent), and byte 15 SHOULD be reserved and set to 0x00 for future use.
When receiving or forwarding *Pong* descriptors, the Descriptor Id field MUST NOT be modified, whatever its value.

3) **Responding to incoming "direct" and "browsing" Ping requests:**
Each servent SHOULD respond (at least once for each connected remote servent) with a valid *Pong* answer to an incoming "direct" *Ping* request with *TTL*=1 and *Hops*=0. To allow the implementation of large *Pong* caches, they SHOULD also advertize (at least once) with *Pong* the list of their currently connected (or recently cached) accessible neighbor servents in reply to an incoming "browsing" *Ping* request (with *TTL*=2 and *Hops*=0).

3.2.2.2. Port numbers in standard Pong descriptors

1) **Standard and default Port numbers:**
Even though Gnutella servents traditionally use TCP *Port* number 6346 *by default* for incoming Gnutella connections,

this is NOT a requirement. There's no "standard" port number defined and servents may use whatever valid port number between 1 and 65535 they wish for Gnutella TCP connections to reach the servent.

2) **Gnutella Port number and download Port number:**
The *Port* number advertized in Pong descriptors MAY be different from the port number advertized in *QueryHits* replies to enable download requests. Incoming Gnutella connections MAY as well assign the same TCP port for incoming HTTP connections used by download requests.

3) **Non null Port numbers:**
A non null Port number indicates support by the servent for incoming Gnutella TCP connections. Most servents SHOULD provide this field with a default value for the local host, unless the servent is discovered to be firewalled or manually configured to use an acceptable port.
The 0.4 protocol currently does not specify any procedure to check that the advertized TCP port number is accessible from other servents or to discover which port is directed to the local host by the firewall or router.

4) **Null Port numbers:**
However, if the servent runs on a host whose whose local IP address is on private LAN and the currently connected Host is on another subnetwork or on Internet, and if Port number has not been explicitly configured by the user for that network interface, it is expected that the default *Port* number will not be accessible; in that case it MAY be preconfigured to 0.
Servents that receive a null Port number in an incoming *Pong* SHOULD discard this Descriptor and not forward it to other servents, as it indicates that direct Gnutella connection with TCP to the sending host is not possible.
Note however that the presence of *Pong Data* may change this behavior, as it may provide alternate transport protocols *(apart from TCP)* to connect to the "firewalled" servent. Such extension is out of scope of the current specification.

5) **Firewalled servents:**
A firewalled servent that cannot accept incoming TCP connections SHOULD set the *Port* field to 0, if a Pong has to be sent in reply to a "direct" Ping whose *TTL*=1 and *Hops*=0 (this will avoid unsuccessful attempts by other remote servents to connect to the firewalled servent). The neighbor servents that accepted the incoming connection from a firewalled servent and that receives such a *Pong* is then informed explicitly that the connected servent does not accept incoming TCP connections, so they need not later advertize this firewalled servent in the list of servents currently connected, when answering to an incoming "browsing" *Ping* (i.e. with *TTL*=2).
Note however that the presence of *Pong Data* may change this behavior, as it may provide alternate transport protocols *(apart from TCP)* to connect to the "firewalled" servent.

Such extension is out of scope of the current specification.

3.2.2.3. IPv4 Addresses in standard Pong descriptors

1) **Unroutable IPv4 Addresses:**

When sending a *Pong* descriptor reporting an IPv4 address to a remote servent, the reported address SHOULD be one that can be safely connected and is accessible to by this servent.

For example, if the remote servent to which a *Pong* descriptor is sent is connected via the global Internet, the local servent SHOULD NOT give him any private network *Address*es (i.e. in the 10/8, 172.16/12, 192.168/16 IPv4 address blocks) that are not routable via the Internet, and SHOULD set this field to 0. The same rule SHOULD also apply if both servents are connected via distinct private networks.

If a servent receives such *Pong*s with unroutable *Address* from remote servents on Internet, the address in these Pongs SHOULD be ignored (as if it was set to 0) even if it matches an accessible address on a local private network, because the reported servents are not accessible or MAY conflict with other hosts on a local private network.

When forwarding those Pongs to any other servent, the unroutable Address field MAY be forced to 0, for example if the other servent is connected from a local private network.

2) **Firewalled servents:**

A firewalled servent that cannot accept incoming IPv4 connections from the network (for example the Internet) to which it wants to send *Pong*s SHOULD set this field to 0, if a *Pong* has to be sent in reply to a "direct" *Ping* request whose *TTL*=1 and *Hops*=0 (this will avoid unsuccessful attempts by other remote servents to connect to the firewalled servent).

Then, the neighbor servents that accepted the incoming connection from a firewalled servent and that receives such a *Pong* is explicitly informed that the connected servent does not accept incoming connections at an accessible IPv4 address, so they NEED NOT later advertize this firewalled servent in the list of servents currently connected, when answering to an incoming "browsing" Ping (i.e. with *TTL+Hops*=2).

Note however that the presence of *Pong Data* extension field may change this behavior, as *Pong Data* may provide alternate host addresses (apart from IPv4) to connect to the servent. Such extension is not described in the current specification.

3.2.3. Query (0x80) Descriptor Payload

Fields	Minimum	Search	NUL (0x00)	*(Optional)*

	Speed	Criteria String	Terminator	*Query Data*
Byte offset	0...1	2...N	N+1	*N+2...L-1*

Port

The TCP port number on which the responding host can accept incoming Gnutella connections.

Minimum Speed

The minimum speed (in kbits/second) of servents that should respond to this message. A servent receiving a *Query* descriptor with a *Minimum Speed* field of *n* kb/s SHOULD only respond with a *QueryHits* if it is able to communicate at a speed >= *n* kb/s.

Here are some hints on how this field MAY be set in Query descriptors:

0 = will send results regardless of available upload speed (and even if there's no available upload slot);
1 = accept any result that can be transferred at a guaranteed minimum of 1.5 kbps mean speed (i.e. modem servents SHOULD NOT report hits if they don't have available upload slots or as long as this would break a guarantee offered to other downloaders, unless they implement a reliable queueing system);
2 to 32767 = (currently available downlink bandwidth) * 70%.
32768 to 65535 = SHOULD NOT be used in 0.4 Query descriptors. Unaware legacy servents that receive such Query descriptors will VERY PROBABLY never return *QueryHits*.

To determine which uplink speed can be guaranteed, the replying upload servent MAY compare the *Minimum Speed* field value *n* of the Query to:

min[(maximum uplink bandwidth) * 70%, (total unused uplink bandwidth)]
/ [(uploads in progress) + 2]

The restricted range of valid values of the *Speed* field in QueryHits descriptors (below 32768), and its typical value is now considered informative, and more useful information can now be transported in this *Minimum Speed* field of Query payloads (See the description of the *Speed* field in the QueryHits in section 3.2.4).

Search Criteria String

A NUL (i.e. 0x00) terminated search string. The maximum length of this string is bounded by the *Payload Length* field of the descriptor header.
It SHOULD use an ASCII-compatible encoding and charset. In this version of the protocol, no encoding was specified, but most servents use the ISO-8859-1 character set, but other encodings such as UTF-8 MAY also be used (possibly

in conjonction with Query Data), as well as other international character sets (ISO-8859-*, KOI-8, S-JIS, Big5, ...).
It MAY consist in an ASCII SPACE (0x20 = 32) separated list of search keywords, that MAY optionally be terminated by one or more filename extensions (after an ASCII dot, 0x2e=46).
For interoperability with future revisions of the 0.4 protocol, the search Criteria field SHOULD NOT use the ASCII FS separator (0x1c = 28).
Also a Query that contains an empty Search Criteria is valid if it is followed by the required NUL terminator and by some Query Data (so the Payload Length cannot be lower than 4 bytes).

Query Data

This is an optional field, of variable length, it is reserved for extensions of the current version of the protocol. Its maximum length is bounded by the *Payload Length* field of the header.
When used, this field SHOULD not be excessively large and agreed upon with other Gnutella servent implementors, as this field MAY be specified in a further specification of the protocol.
Popular servent implementations use this field to specify extended search requests based on meta-data, encoded as a NUL-terminated string containing additional XML formated search criterias. Other extensions may follow this second NUL byte.
Servents SHOULD then forward these optional extensions when they are present.

3.2.4. QueryHits (0x81) Descriptor Payload

Fields	Number of Hits	Port	IP Address	Speed	Result Set	*Optional QHD Data*	Servent Identifier
Byte offset	0	1...2	3...6	7...10	11…10+N	*11+N...L-17*	L-16...L-1

Number of Hits

The number of query hits in the *Result Set* field (see below).

Port

The port number on which the responding host can accept incoming connections.

IP Address

The IPv4 address of the responding host.
This field is in **big-endian** format.

Speed

The maximum upload speed (in kilobits/second or bits/millisecond, between 0 and 32767) of the responding host.
This legacy semantic is deprecating, as this does not

guarantee a good effective upload speed, as this depends on the effective workload of the host, and its number of available upload slots (so the speed is only informative and servents should not consider it). As the typical value of this field is typically low, the most significant bit of this 16-bit field (sent in little-endian order, as all other Gnutella messages) is now used as a case selector, that allows sending more useful information:

Legacy format	0	Maximum upload speed in kbit/s														
Extended format	1	Firewalled indicator	XML meta-data	Unassigned bits, set to 0					Reserved bits, set to 0							
Bit in *Speed* fi eld	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Bit in byte	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0
Byte offset	1								0							

When the extended format is used (bit 15 set to 1), the other fields are defined as below:

Firewalled indicator

This bit is set to 1 in *Query* and *QueryHits* payloads, to indicate that the host emitting the message is firewalled. When both the initial Query source and the receiver are firewalled, the responding servent should not respond to the *Query*, as its *QueryHits* won't be accessible, even with the *Push* mechanism (see section 3.2.5)

XML meta-data

This bit is set to 1 in *Query* payloads, to indicate the desired preference to receive extended meta-data for results sent in Query Hits, using the LimeWire XML format. Only new LimeWire servents honor this bit, and other servents implementing XML meta-data in their results, should be changed to honor this bit too. Any servent that can interpret XML meta-data should set this bit to 1 in its *Query* to allow receiving them in the extension field *Optional QHD Data* (see below). Other servents that cannot interpret XML meta-data, or servents that do not want to receive them in *Optional QHD Data* should clear this bit (see Appendix A.2.3).

Unassigned bits

Currently unassigned, reserved for future use. Until then, these bits should be set to 0.

Reserved bits

Currently unassigned, reserved for future indication of a maximum number of hits to return. Until

specification, these bits should be set to 0.

Result Set

A set of responses to the corresponding Query. This set contains *Number of Hits* elements, each with the *Result* structure described below (section 3.2.4.1).
The size of the *Result Set* field (and of each individual *Result* structure it contains) is bounded by the size of the *Payload Length* field in the descriptor header and by the *Number of Hits* field, minus the size of the required *Servent Identifier* field at end of this payload.

Optional QHD Data

This is an optional field (the Extended QueryHits Data) of variable length, it is reserved for extensions of the current version of the protocol. Its maximum length is bounded by the Payload Length field of the header. (see section 3.2.4.3) When used, this field SHOULD not be excessively large and agreed upon with other Gnutella servent implementors, as this field MAY be specified in a further specification of the protocol.
Some servents use this field to give other collected information about the Query or about the responding host. Servents SHOULD then forward these optional extensions when they are present. (see Annexes)

Servent Identifier

This 16-byte string uniquely identifies the responding servent on the network. This is typically some function of the servent's network address.
The *Servent Identifier* is instrumental in the operation of the *Push* descriptor (see below).

QueryHits descriptors are only sent in response to an incoming *Query* descriptor. A servent should only reply to a *Query* with a *QueryHits* descriptor if it contains data that strictly meets the *Query Search Criteria*.

A QueryHits descriptor SHOULD be initially generated with *Hops*=0 and the *TTL* field equal to the number of *Hops* traversed by the *Query* descriptor for which it is replies.

The *Descriptor Id* field in the descriptor header of the *QueryHits* should contain the same value as that of the associated *Query* descriptor. This allows a servent to identify the *QueryHits* descriptors associated with *Query* descriptors it generated.

3.2.4.1. Result Structure

Fields	File Index	File Size	Shared File Name	NUL (0x0) Terminator	*Optional Result Data*	NUL (0x0) Terminator
Byte offset	0...3	4...7	8...7+K	8+K	*9+K...R-2*	R-1

File Index
A number, assigned by the responding host, which is used to uniquely identify the file matching the corresponding query.

File Size
The size (in bytes) of the file whose index is *File Index*.

Shared File Name
The nul (i.e. 0x0000) terminated shared name of the file whose index is *File Index*.

Optional Result Data
Nul (i.e. 0x0000) terminated data about the file whose index is *File Index*. Some servents MAY use this field to return meta-data, encoded with XML. Care MUST be taken not to include any NUL byte in this field.
A text-only extension (some as XML data) in this field SHOULD be terminated by an ASCII File Separator (FS, 0x1c=28) if there is another extension after it.
When this field is not used, the second NUL terminator MUST still be present.

3.2.4.2. Total Payload length of QueryHits descriptors.

The *QueryHits* descriptor, with its complex structure, is the one which may have the longest payload. For efficiency and to allow more concurrent requests, a servent that receives a *Query* SHOULD limit the volume of the *QueryHits* descriptor it sends as a reply. When many hits are detected, servents MAY and SHOULD divide it in reasonable subsets, with a delay between each QueryHits descriptor sent back to the requester.

Any QueryHits descriptor SHOULD NOT need more than 4 seconds to transmit at an average speed per connection of 4kbps, because the servent needs to be able to reply to other incoming requests from its connected neighbors in a timely way. In practice, this limits the total descriptor size to 2KB, unless more uplink bandwidth is available, and if there's agreement (or negociation at connection time) about the maximum descriptor size that can be used between neighbor servents. This can only be guaranteed if replying with a *TTL*=1 descriptor, which explicitly won't need to be routed across relaying servents, and which has greater priority than other descriptors with larger *TTL* values.

When a servent replies to an incoming *Query* descriptor with *Hops*>0, the *QueryHits* descriptor with *TTL*>1 will return back to the initial servent that sent the *Query*, using the same connection path that was used when receiving the *Query*. Most neighbor servents will forward incoming *QueryHits* descriptors (with *TTL*>1) without breaking them into their individual components. Buffer size limits in relaying servents MAY impact the intended descriptor, as a relaying agent MAY drop a too long QueryHits descriptor.

All servents SHOULD be able to route *QueryHits* descriptors with

total size (including the descriptor header) up to 2KB. And servents SHOULD NOT generate any *QueryHits* descriptor with more than 64KB total size, unless there's mutual agreement that such large descriptors can be safely exchanged. Between these figures, the maximum descriptor size CAN be reduced in an order of magnitude proportional to the increase of the *TTL+Hops* value.

3.2.4.3. QHD Data and Result Data Extensions

The *QueryHits* descriptor allows two kinds of extensions, either per Result or for the whole Result-Set. The choice of placement of these extensions (and their encoding and semantics) is not defined in this document; it's up to the implementers of servents to define and test them, however several things should be noted:

Important notice: This section gives some guidelines, but the "SHOULD" and "MAY" words found here are still being debated, particularly for the semantics of cachable extensions that could be splitted or merged by future Query-caching relaying servents.

1) **Extensions encoding:**

There MAY be several Result Data extensions for the same *Result* file. And there may be several QHD Data extensions for the same QueryHits descriptor.
Some extensions are also versatile, i.e. they MAY be used in descriptors with different Payload type.
So each extension MUST start with a distinct identifiable sequence to recognize its type. The following paragraphs give examples for interoperability of some existing extensions.

1.a) **XML extensions**

start with a ASCII "<" character (0x3c=60), which is part of its actual content and can be a comment, a document type declaration, or the beginning of the document element; an XML extension is terminated by a ASCII FS character if followed by another extension.
Standard XML data uses the UTF-8 encoding by default, but MAY use other explicit encodings. For interoperability, implementers of XML extensions SHOULD produce XML data using an explicit default target namespace and/or a distinctive document element name. Full XML conformance is not required, and relaying servents don't need to validate them.

1.b) **URN extensions**

start with a ASCII lowercase "u" letter (0x75=117), part of its value, and terminated by a ASCII FS, if followed by another extension.

1.c) **Sets of *GGEP* binary-delimited extensions**

(not specified here) are introduced by a *magic* byte (0xc3=195), and each extension in the set contains a *length* indicator and an extension-specific signature;

however no byte in the *GGEP* binary-delimited extension may be NUL (0x00) when encoded within a *Result Data* field (this MAY use a special binary encoding). The whole set of GGEP-compatible binary extensions is terminated by a ASCII FS if followed by another extension.

1.d) ***BearShareTrailer*-type binary extensions**
(see Appendix 1) start with a vendor-specific Identifier of 4 ASCII characters (it SHOULD not start by a "<", "u" or 0xc3 byte) and specify their internal data length. Such binary extensions are not designed to be used in a *Result Data* extension field, but only in *QHD Data* extension. For newer applications, *GGEP*-style extensions SHOULD be preferred.

2) **When to use *QHD Data* extensions:**

Servents MAY need to split a large incoming result-set into several distinct QueryHits descriptors, each one transmitted after a time delay, to better manage its outgoing bandwidth and allow responding to other requests.
When it needs to do so, it MAY transmit the QHD Data separately in an empty or tiny result set, or MAY have to repeat the *QHD Data* in each QueryHits descriptors.
Servents that send large *QHD Data* SHOULD design their extension in such a way that this data MAY be transmitted separately (however with the same responding Servent Identifier field), or so that this data MAY be repeated in multiple Query Hits.
So any meta-data associated with a single file would better not be within this *QHD Data* extension field, and the *QHD Data* will only be best used either with single-file *Results Set*, or to transport small servent-related information.
Also *QHD data* cannot be parsed without first receiving and parsing the *Result Set*, because there's no length indicator for the *Result Set*: each Result structure must be scanned while counting them, until *Number of Hits* have been scanned. For faster routing purpose, a servent MAY also need to limit the *Number of Hits* allowed in the same Result Set.
Servents SHOULD also avoid transmitting *QueryHit* descriptors with empty *Result Set* in order to send only *QHD Data* extensions, as some legacy servents MAY discard such empty descriptors.
The 0.4 protocol does not specify however that an empty Result Set is invalid. So, new servents SHOULD accept and forward *QueryHits* descriptors containing an empty *Result Set* if it contains *QHD Data* extension.
Finally, all servents SHOULD discard *QueryHits* descriptors with both empty *Result Set* (i.e. *Number of Hits*=0) and no *QHD Data extension* (i.e. *PayLoad Length*<=27).

3) When to use Result Data extensions:

To avoid such split of related information, meta-data can be encoded, along with the Result with which it is related, within the *Result Data* extension. However, pure binary

format for these extensions is not possible as *Result Data* extensions MUST NOT contain NUL bytes; additionally it SHOULD NOT contain the ASCII File Separator (FS, 0x1c = 28) used to terminate text-only extensions with no explicit length. This MAY then require a less efficient (larger) encoding for such meta data within the *Result Data* extension field of a Result structure.

Using a *Result Set* with several combined hits saves a little output bandwidth when we compare it to the bandwidth needed when using an equivalent splitted *Result Set*, because of the headers overhead. However servents SHOULD avoid using descriptors with excessive length, as it may cause buffering problems in remote servents.

4) When to anticipate splitted QueryHits:

If a QueryHits extension is large then it SHOULD be carefully designed to differentiate servent-related information from files-specific meta-data.

Servent-related information SHOULD not be sent within multiple *QueryHits* descriptor associated with the same *Query* request (identified by the matching *Descriptor ID* field in the descriptor header), but only with the first Result Set for that *Query*. It SHOULD be encoded as a *QHD Data* extension, and this first *Result Set* MAY need to be reduced to contain the smallest *Result* structures.

Large file-related meta-data MAY be encoded as a *QHD Data* extension instead of a *Result Data* extension to allow better encoding. In such a case, the *Result Set* MAY need to be reduced to contain only one *Result* structure.

A *QHD Data* extension MAY be designed to include a vector of file-related meta-data, one for each file of the *Result Set*. However as a Result Set MAY be splitted by relaying agents, with *QHD Data* extensions replicated in each *QueryHits* descriptor, it would be difficult to reassociate the meta-data with the correct file. In that case, the extension may include the *File Id* within each element of the vector encoded in the *QHD Data* extension.

Until the semantics of splitting (or merging) a *Result Set* are standardized in a future version of this specification, servents need to be carefully tested with other popular implementations, to determine the appropriate policy, as it MAY break the behavior of an existing extension (for example if *QueryHits* are digitally signed)

3.2.5. Push (0x40) Descriptor Payload

Fields	Servent Identifier	File Index	IP Address	Port	*Optional Push Data*
Byte offset	0...15	16...19	20...23	24...25	*26...L-1*

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 should set this field to the *Servent Identifier* returned in the corresponding *QueryHits* descriptor. This allows the recipient of a *Push* request to determine whether or not it is the target of that request.

File Index

The index uniquely identifying the file to be pushed from the target servent. The servent initiating the *Push* request should set this field to the value of one of the *File Index* fields from the *Result Set* field in the corresponding *QueryHits* descriptor.

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.

Port

The port to which the file with index *File Index* should be pushed.

Optional Push Data

This is an optional field of variable length, it is reserved for extensions of the current version of the protocol, to give identifying information about the content to push, or routing and authenticating information collected from previous *QueryHits* and/or *Pong* descriptor. Its maximum length is bounded by the *Payload Length* field of the descriptor header.

When used, this field SHOULD not be excessively large and agreed upon with other Gnutella servent implementors, as this field MAY be specified in a further specification of the protocol.

Servents SHOULD then forward these optional extensions when they are present.

A servent may send a *Push* descriptor if it receives a *QueryHit* descriptor from a servent that doesn't support incoming connections. This might occur when the servent sending the *QueryHits* descriptor is behind a firewall. When a servent receives a *Push* descriptor, it may act upon the push request if and only if the *Servent Identifier* field contains the value of its servent identifier.

The *Descriptor Id* field in the Descriptor Header of the Push descriptor should not contain the same value as that of the associated *QueryHits* descriptor, but should contain a new value generated by the servent's *Descriptor_Id* generation algorithm. See the section below entitled "Firewalled Servents" for further details on the Push process.

3.2.6. QRP (0x30) Extension Descriptor Payload

Fields	*Quary Routing Table Data*
Byte offset	*0...L-1*

Query Routing Table Data

This is a required field consisting in bytes of variable length, it is reserved for extensions of the current version of the protocol, to send compact information about files shared by a servent, in order for the recipient to filter incoming Queries.
This field can be large, but the descriptor should be compacted with an algorithm not specified in this document..

This descriptor was not specified in the original 0.4 protocol. Implementing it in servents is optional (but sending it is required to implement the "Leaf node" mode specified in the UltraPeer extension. It should be sent only to servents implementing the UltraPeer protocol, as indicated in their connection headers. Non-QRP aware servents MAY safely ignore this descriptor, as it is completely compatible with all non *QRP*-aware 0.4 servents that don't use it.

The routing of this descriptor is not defined in this document. Its presence in a reception flow indicates that the recipient should support the QRP mechanism, most probably to implement the UltraPeer topology extension. Its occurrence in a flow sent by a given servent should be paced according to the QRP protocol extension that defines it. Generally, this message is not intended to be dropped by the recipient. So receiving it while it was not solicited indicates that the servent does not comply strictly to this specification but already implements a part of the QRP extension, but does not comply to its specification.

3.2.7. Bye (0x02) Extension Descriptor Payload

Fields	*Optional Bye Data*
Byte offset	*0...L-1*

Optional Bye Data

This is an optional field consisting in bytes of variable length, it is reserved for extensions of the current version of the protocol, to specify filters about expected Pong replies. Its maximum length is bounded by the Payload Length field of the header.
When used, this field SHOULD be small and agreed upon with other Gnutella servent implementors, as this field MAY be specified in a further specification of the protocol.

This descriptor was not specified in the original 0.4 protocol. Implementing it in servents is optional. Servents MAY safely ignore this descriptor, as it is completely compatible with all non *Bye*-aware 0.4 servents.

However a *Bye*-aware servent MUST set *TTL*=1 and *Hops*=0 when sending this descriptor, then it SHOULD NOT send or forward any other descriptor on the same connection path; instead it MAY wait for about 30 seconds that the connection closes (if timeout elapses, it SHOULD close the connection). During that period, the servent MAY ignore all other incoming descriptors coming from the same connection path (with the exception of another incoming *Bye* Descriptor which MAY be interpreted). The semantic of an sending a *Bye* descriptor with *Hops*<>0 is unknown and not defined in this document.

On reception, a *Bye*-aware servent MUST NOT forward this message; it MAY interpret the Payload to take further actions, but it SHOULD disconnect immediately from the servent which sent this descriptor. The content of the Payload is not specified in this version of the protocol (it will typically contain a NUL terminated status line that gives the reason why a servent will be disconnected, and other *Optional Bye Data* extensions).

3.2.7. Open-Vendor (0x31) Extension Descriptor Payload

Fields	Vendor ID	Sub-Selector	*Optional Sub-Selector Version*	*Optional Vendor Sub-Selector Data*
Byte offset	0...3	4...5	*6...7*	*6...L-1*

Vendor ID
: Case insensitive sequence of 4 characters, identifying the vendor who has authority on the descriptor format and its definition. *Vendor ID* values are similar to those used in *QueryHits (See Appendix 1)*. The all-zero value is reserved for Vendor support requests and answers. See below.

Sub-Selector
: A little-endian 16-bit value specifying a distinct message type defined by that vendor. The 0xFFFF and 0xFFFE values for the *Sub-Selector* field are reserved for feature request and answers. See below.

Optional Sub-Selector Version
: A little-endian 16-bit value specifying a variant for the distinct message type defined by that vendor. The 0x0000 value is assumed if absent. Some Sub-Selectors will be versioned and some won't. The value 0x0001 represents version 0.1.

Optional Vendor Sub-Selector Data
: This is an optional field consisting in bytes of variable length. Its format depends on the *Vendor ID*, *Sub-Selector* and *Optional Sub-Selector Version* fields. Its maximum length is bounded by the *Payload Length* field of the header.

This descriptor was not specified in the original 0.4 protocol.

Implementing it in servents is optional. It allows servents to send experimental messages, and test their scalability and routing strategies for networking enhancements without breaking other existing servent implementations.

However servents that implement this descriptor SHOULD also implement the Open-Vendor Feature request/answer 0.1 mechanism. See below.

Non-aware servents MAY safely ignore this descriptor, as it should be completely compatible with all non-Vendor aware 0.4 servents.

However a *Open-Vendor*-aware servent SHOULD set *TTL*=1 and *Hops*=0 when sending this descriptor. In that case, the *Descriptor ID* field may be used for other usage than identifying the uniqueness of the originator. The semantic of sending a *Open-Vendor* descriptor with *TTL*>1, or forwarding it with *Hops*<>0 is unknown and not defined in this document.

The maximum size of this Descriptor should be below 20 KB for routing purpose, and MUST NOT exceed 64KB with *TTL*=1 and *Hops*=0.

On reception, a non-aware servent MUST NOT blindly forward this descriptor; it MAY interpret the Payload to take further actions. The content of the Payload is not specified in this version of the protocol, as it is vendor-specific, and may change over time.

Querying the list if *Vendor ID* supported in *Open-Vendor* descriptors:

> One servent A can query which *Vendor IDs* the remote servent B support: it sends an *Open-Vendor* descriptor with the *Vendor ID* field set to all-zeroes, and sets the *Sub-Selector* field to 0xFFFF and *Version* field to 0x0001. If B supports some *Open-Vendor* descriptors, it will answer by sending back another *Open-Vendor* descriptor with the *Vendor ID* field set to all-zeroes, and the *Sub-Selector* field set to 0xFFFE, and the *Optional Sub-Selector version* field set to 0x0001; the *Optional Vendor Sub-Selector Data* field will contain the list of *Vendor IDs* supported.

Querying if one or more specific *Vendor ID* are supported in *Open-Vendor* descriptors:

> This uses the same mechanism, unless that the servent A will insert one or more *Vendor ID* values in the *Optional Vendor Sub-Selector Data* field. The servent B will reply by listing only those Vendor IDs values that are supported in the requested set. If B does not support any of these value, it can explicitly reply with an empty list of *Vendor ID* values in the *Optional Vendor Sub-Selector Data* field of its answer. This type of request allows restricting the volume of data exchanged between servents because the servent B may support a large set of Vendor-specific extensions.

Querying the list of *Open-Vendor Sub-selectors* supported for a

specific *Vendor ID*:

One servent A can query which *Sub-Selectors* the remote servent B support: it sends an *Open-Vendor* descriptor with the *Vendor ID* field set according to the Sub-Selectors to query, and sets the *Sub-Selector* field to 0xFFFF and *Version* field to 0x0001. If B supports *Open-Vendor* descriptors for that *Vendor ID*, it will answer by sending back another *Open-Vendor* descriptor with that same *Vendor ID*, and the *Sub-Selector* field set to 0xFFFE, and the *Optional Sub-Selector version* field set to 0X0001; the *Optional Vendor Sub-Selector Data* field will contain the list of Sub-Selectors supported with that Vendor ID.
A more precise query that takes version fields or identification fields into account may be used with *Sub-Selector Version* field set to 0x0002, in the 0xFFFF "Feature Query" *Sub-Selector*, or in the 0xFFFE "Feature Answer" *Sub-Selector*. In that case, each *Sub-Selector* value listed in the *Optional Vendor Sub-Selector Data* field will be followed by a length byte and the version information.

3.2.8. Standard-Vendor (0x32) Extension Descriptor Payload

Fields	Vendor ID	Sub-Selector	*Optional Sub-Selector Version*	*Optional Vendor Sub-Selector Data*
Byte offset	0...3	4...5	*6...7*	*6...L-1*

The structure of this descriptor is completely identical to the structure of the 0x31 descriptor type. In fact it is highly recommended that servents that implement any 0x32 descriptor also accepts receiving its 0x31 experimental variant with exactly the same Payload. The **feature query** mechanism can also be used to see if a legacy servent supports the approved 0x32 variant. However a servent that implements the approved 0x32 variant should no more reply with the experimental 0x31 variant of the descriptor.

This descriptor was not specified in the original 0.4 protocol. Implementing it in servents is optional. It allows servents to send experimental vendor-specific messages, for networking enhancements without breaking other existing servent implementations, but it restricts its definition to a stable and documented specification. Experimental *Open-Vendor* 0x31 descriptors may be safely ignored, but detecting a *Standard-Vendor* 0x32 message gives a hint to the implementor about which *Open-Vendor* descriptor they should monitor and implement in their next releases as per the available specification.

This descriptor may have special routing strategies. In that case, this descriptor MUST be sent with *TTL*=1 and *Hops*=0, and its

unique 128-bit *Descriptor ID* MAY be used for other purpose. If the *Standard-Vendor* descriptor uses a standard forwarding strategy, it should include a unique 128-bit *Descriptor ID* which MUST be preserved while incrementing the *Hops* field and decrementing the *TTL* field. Implementing effectively an *Standard-Vendor* descriptor MAY require complex caching strategies. For testing purposes, and to limit the impact of possible bugs, all tests SHOULD be performed using the Experimental *Open-Vendor* descriptors, so that it won't harm other conforming servents.

Only when the implementation passes the specification compliance tests with other major servent implementations present on the network that implement this *Open-Vendor* message, the implementor SHOULD replace any occurence of experimental 0x31 descriptors for that *Vendor ID* and *Sub-Selector* by 0x32 descriptors in requests and in replies to incoming approved 0x32 descriptors. Going to the *Standard-Vendor* state will allow more reachability of this *Open-Vendor* descriptor on the network.

When answering to an incoming *Open-Vendor* 0x31 descriptor, *Standard-Vendor* 0x32 descriptors MUST NOT be used, whatever its content, unless the incoming *Vendor ID* matches the receiving servent implementation and the receiving servent is fully compliant to the *Standard-Vendor* descriptor specification.

When answering to an incoming *Standard-Vendor* 0x32 descriptor, *Open-Vendor* 0x31 descriptors SHOULD NOT be used, other *Standard-Vendor* 0x32 descriptors SHOULD be used instead. Other type of answers MAY include other descriptors such as *Ping*, *Pong*, *Query*, *QueryHits*, *Push* and *Bye*, or any other action defined in the *Standard-Vendor* descriptor.

---

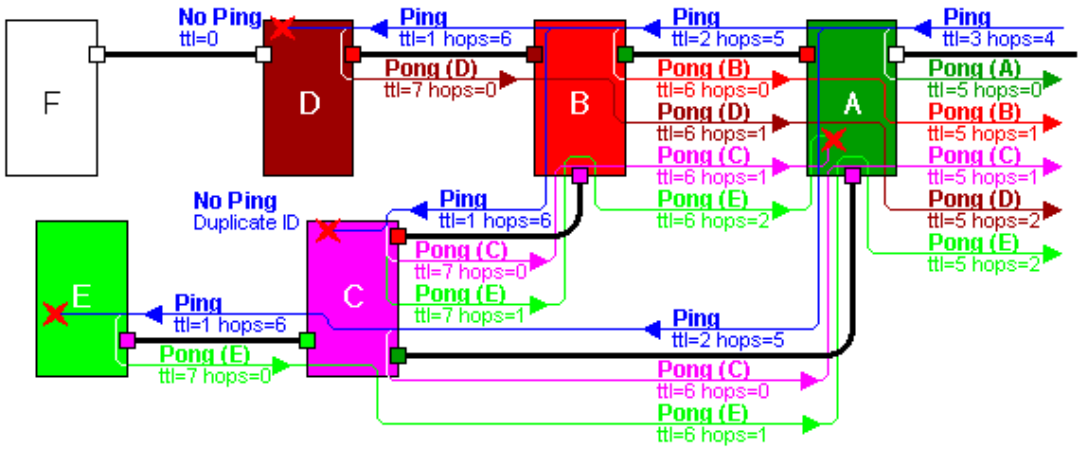
4. Descriptor Routing

The peer-to-peer nature of the Gnutella network REQUIRES servents to route network traffic (queries, query replies, push requests, etc.) appropriately. A well-behaved Gnutella servent MUST route protocol descriptors according to the following rules:

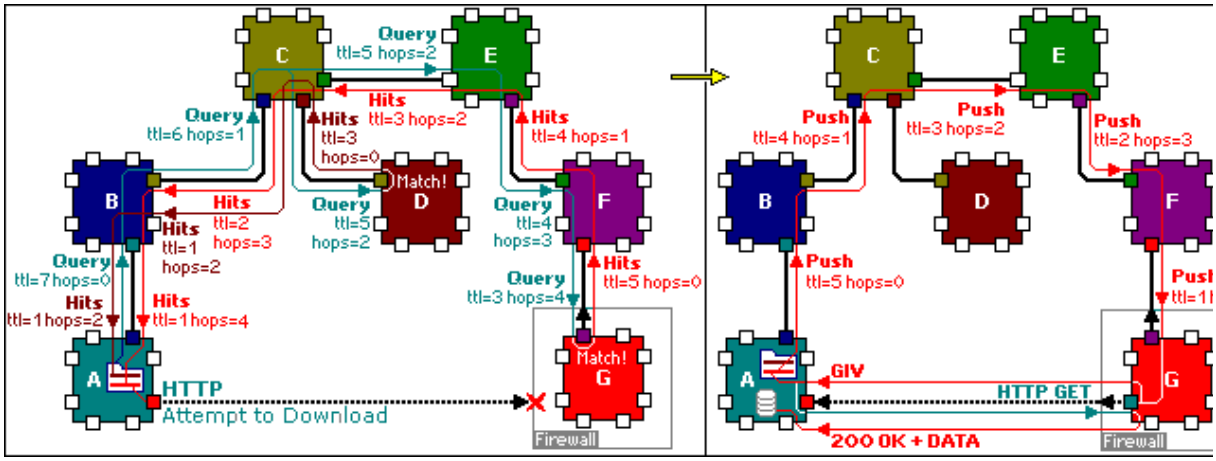
1. **Pong** descriptors MAY ONLY be sent along the **same path** that carried the incoming **Ping** descriptor. This ensures that only those servents that routed the Ping descriptor will see the Pong descriptor in response. A servent that receives a Pong descriptor with Descriptor ID = *n*, but has not seen a Ping descriptor with Descriptor ID = *n* SHOULD remove the Pong descriptor from the network and not forward it to any connected servent.
2. **QueryHit** descriptors MAY ONLY be sent along the **same path** that carried the incoming **Query** descriptor. This ensures that only those servents that routed the Query descriptor will see the QueryHit descriptor in response. A servent that receives a QueryHit descriptor with Descriptor ID = *n*, but has not seen a Query descriptor with Descriptor ID = *n* SHOULD remove the QueryHit descriptor from the network.
3. **Push** descriptors MAY ONLY be sent along the **same path** that carried the incoming **QueryHit** descriptor. This ensures that only those servents that routed the QueryHit

descriptor will see the Push descriptor. A servent that receives a Push descriptor with Servent Identifier = *n*, but has not seen a QueryHit descriptor with Servent Identifier = *n* SHOULD remove the Push descriptor from the network. Push descriptors MUST be routed by Servent Identifier, not by Descriptor Id *(this id is not related to the Descriptor Id of any Query or Query Hit descriptor, it is used only to identify the push request when the uploader servent will try to connect to the downloader, so it MAY be related to the local servent GUID that initiates the Push descriptor)*.

4. A servent SHOULD **forward** incoming Ping and Query descriptors to ALL of its directly connected servents, **except** the one that delivered the incoming Ping or Query.
5. A servent MUST decrement a descriptor header's **TTL** field, and increment its **Hops** field, **before** it forwards the descriptor to any directly connected servent. If, after decrementing the header's TTL field, the TTL field is found to be zero, the descriptor is not forwarded along any connection.
6. A servent receiving a descriptor with the **same Payload Descriptor and Descriptor ID** as one it has received before, SHOULD attempt to avoid forwarding the descriptor to any connected servent. Its intended recipients have already received such a descriptor, and sending it again merely wastes network bandwidth.



Example 1. Ping/Pong routing, with duplicate descriptors filtering



Example 2. Query/QueryHit/Push routing

5. File Downloads

Once a servent receives a QueryHit descriptor, it may initiate the direct download of one of the files described by the descriptor'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.

> Important note: This document is not normative as regard to the HTTP protocol. It only defines the **minimum HTTP support** REQUIRED to implement compliant servents. Any compliant HTTP/1.0 or HTTP/1.1 client or server MAY be used. For further reference, please consult:
> [1] **RFC 2616** "Hypertext Transport Protocol - HTTP/1.1", by Irvine, Gettys, Compaq, W3C, MIT & al. (2nd release, June 1999), Standard Track, edited by W3C, available in HTML format on http://www.w3.org/Protocols/rfc2616/rfc2616.html
> *This version obsoletes the previous:*
> *[2] RFC 2068 "Hypertext Transport Protocol - HTTP/1.1", by Fielding, Irvine, MIT & al. (1st release, January 1997), Proposed Standard, edited by IETF.*
> [3]**RFC 1945** "Hypertext Transport Protocol - HTTP/1.0", by Berners-Lee, Fielding, Frystyk & MIT/LCS (May 1996), Informational, edited by IESG, available in ASCII text format on http://www.ietf.org/rfc/rfc1945.txt

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

```
GET /get/<File Index>/<File Name>/ HTTP/1.0\r\n
User-Agent: Gnutella/0.4\r\n (3)
Range: bytes=<Start Offset>-\r\n
Connection: Keep-Alive\r\n
\r\n
```

where *<File Index>* and *<File Name>* are one of the File Index/File Name pairs from a *QueryHits* descriptor's *Result Set*. For example, if the *Result Set* from a *QueryHits* descriptor contained the following *Result* entry:

File Index	2468
File Size	4356789
File Name	Foobar.mp3\x00
Result Data	\x00

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

```
GET /get/2468/Foobar.mp3/ HTTP/1.0\r\n
User-Agent: Gnutella/0.4\r\n (3)
Range: bytes=0-\r\n
Connection: Keep-Alive\r\n
\r\n
```

> *Note that some characters (such as spaces) in the filename may need to be encoded with %hh hexadecimal sequences to comply with the URI standard referred by the HTTP protocol standard.*
>
> *Note also that the incoming HTTP version MAY be 1.1 instead of 1.0. Support for HTTP/1.1 is not mandatory within servents, which can still respond with the*

HTTP/1.0 protocol.

The "Connection: Keep-Alive" header MAY also be omitted in HTTP/1.0, but SHOULD be present in HTTP/1.1. The "Range:" header MAY also be omitted if the request specifies the full file content. Note also the double CRLF sequence required at end of headers, when emitting and parsing HTTP request headers.

The server receiving this download request responds with HTTP 1.0 compliant headers such as:

```
HTTP/1.0 200 OK\r\n
Server: Gnutella/0.4\r\n (3)
Content-Type: application/binary\r\n
Content-Length: 4356789\r\n
\r\n
```

Note that some legacy servents may also respond with a different status line (before the HTTP headers), which does not fully comply to the HTTP 1.0 or 0.9 standard, but to a very deprecated legacy standard, as they may omit the version suffix (this is a consequence of an error in the previous revision of this document) :

```
HTTP 200 OK\r\n
```

Note also that the "Content-Type:" header may also be missing on incoming requests, but all servents SHOULD emit it when uploading files, for compatibility with some HTTP proxies. Note also the double CRLF sequence required at end of headers, when emitting and parsing HTTP answer headers.

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 that if there's no "Content-Length:" specified, the downloading servent has no other choice than HAVING TO read the file data up to the detection of the end of the TCP connection, and abort the download if there's a file size mismatch. So all servents SHALL include this header.

The Gnutella protocol provides support for the HTTP "Range:" parameter, so that interrupted downloads may be resumed at the point at which they terminated.

If the connection does not terminate immediately when the download completes, the downloading servent MAY initiate another "Range:" download by emitting another HTTP GET request for the same file (the uploading servent MAY elect not to honor it and return an HTTP BUSY error).

Once the HTTP transactions are finished or if the uploading server returns any error, the downloading client SHOULD immediately close the connection. A fast uploading servent SHOULD keep the connection open for a limited time before forcing the connection close (this will avoid exhaustion of available socket system control blocks on fast servents that accept many incoming requests).

---

6. 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 descriptor describing the target file. The servent that is the target of the Push request (identified by the Servent Identifier field of the Push descriptor) should, upon receipt of the Push descriptor, attempt to establish a new TCP/IP connection to the requesting servent (identified by the IP Address and Port fields of the Push descriptor). 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.

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 <File Index>:<Servent Identifier>/<File Name>\n\n
```

Where <*File Index*> and <*Servent Identifier*> are the values of the *File Index* and *Servent Identifier* fields respectively from the Push request received, and <*File Name*> is the name of the file in the local file table whose file index number is <*File Index*>. The servent receiving the *GIV* request header (i.e. the Push requester) should extract the <*File Index*> and <*File Name*> fields from the header and construct an *HTTP GET* request of the following form:

```
GET /get/<File Index>/<File Name>/ HTTP/1.0\r\n
User-Agent: Gnutella/0.4\r\n (3)
Range: bytes=0-\r\n
Connection: Keep-Alive\r\n
\r\n
```

The remainder of the file download process is identical to that described in the section entitled "File Downloads" above.

---

(3)

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.

Appendix: Gnutella Protocol Extensions

A.1. Extended Query Hit Descriptor (EQHD)

(Description Updated 03/15/2001)

A.1.1. EQHD common format

First introduced by BearShare v1.3.0, the *Extended QueryHits* Descriptor extends the original Gnutella *QueryHits* descriptor by placing extra data between the last double-nul terminated filename of the *Result Set* and the *Servent Identifier*. An *Extended QueryHit* descriptor (0x81) will have the following payload structure:

Fields	Number of Hits	Port	IP Address	Speed	Result Set	*Optional QHD Data*	Servent Identifier
Byte offset	0	1...2	3...6	7...10	11…N	*N+1...L-17*	L-16...L-1

One way for developers to handle the *QHD Data* extensions is to

1) Be aware that an incoming *QueryHits* descriptor may or may not contain additional data after the *Result Set* and before the *Servent Identifier*. No complete specification exists for the number of bytes that may be present, or their content. Use the *Payload Length* field and count bytes as they are read from the stream to determine whether the extension bytes are present.

2) If they are, read them from the stream, leaving 16 bytes for the *Servent Identifier*.

3) Process the *QueryHit* as usual.

A.1.2. BearShareTrailer EQHD

The *BearShareTrailer* field stored in the *Optional QHD Data* field has the following structure:

Fields	Trailer Vendor Code	Open Data Size	Open Data	*Optional Private Data*	*Optional Signature Binary Data*	*Optional Signature Size*
Byte offset	0...3	4...5	6...6+n-1	*6+n...L-17-N-(s+1)*	*L-17-N-s...L-17-N-1*	*L-17-N*

Vendor Code
Four case-insensitive characters representing a vendor code. (Some recognized vendor codes are listed in section A.1.3).

Open Data Size
Contains the length (in bytes) of the Open Data field.

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

Fields	Flag1	Flag2	*Optional XML Data Length*	*Optional XML Data*	*Optional Vendor Data*
Byte offset	0	1	*2...3*	*4...4+s-1*	*4+s...n-1*

Flag1

flags	*r*	*r*	*r*	setUploadSpeed	setHaveUploaded	setBusy	*r*	flagPush
Bit number	7	6	5	4	3	2	1	0

setUploadSpeed = 1 if and only if the

flagUploadSpeed flag in the *Flags2* field is meaningful.
setHaveUploaded = 1 if and only if the *flagUploaded* flag in the *Flags2* field is meaningful.
setBusy = 1 if and only if the *flagBusy* flag in the *Flags2* field is meaningful.
flagPush = 1 if and only if the servent is firewalled or has not yet accepted an incoming connection.
r = reserved for future use.

Flags2

flags	*r*	*r*	*r*	flagUploadSpeed	flagHaveUploaded	flagBusy	*r*	setPush
Bit number	7	6	5	4	3	2	1	0

flagUploadSpeed = 1 if and only if the *Speed* field of the *QueryHits* descriptor contains the highest average transfer rate (in kilobits per second) of the last 10 uploads.
flagHaveUploaded = 1 if and only if the servent has successfully uploaded at least one file.
flagBusy = 1 if and only if the all of the servent's upload slots are currently full.
setPush = 1 if and only if the *flagPush* flag in the *Flags1* field is meaningful.
r = reserved for future use.

Optional XML Data Length

This little-endian 16-bit field, introduced in LimeWire 1.8 and later, includes the length of the Optional XML Data field that may follow it. It must be set to 0 if no *XML Data* is used but other *Optional Vendor Data* must be used.

Optional XML Data

This optional field contains XML encoded meta-data for the files referred to by the Result structures of the QueryHits. The presence of textual XML can be inferred to by the leading byte which SHOULD be an ASCII "<" for an XML declaration, comment or root element. A leading non-printable ASCII character implies that the field is not encoded with textual XML, but uses some other binary format.

Optional Vendor Data

This optional field other vendor-specific data for the *QueryHits* Descriptor. It may use binary format and it is bound by the Value of the *Open Data Size* field. This vendor-specific field is deprecated in favor of set of GGEP-style extensions in the *Optional Private Data* field described below.

Optional Private Data

Undocumented BearShare-specific data. The length of this field can be determined as follows:

```
<QueryHit Descriptor Payload Size> - ( <Open Data Size> + 4 + 1 ).
```

For interoperability with many newer servents, this field SHOULD NOT start with the magic byte 0xC3 or include the magic bytes sequence (0x1C, 0xC3) that delimits GGEP extensions. This field is deprecated in favor of sets of GGEP extensions, which allow using extensions from multiple vendors, stored in this Private Data field. In that case, the set of GGEP extension is self delimited, and strictly bound to the above size limit.

Optional Signature Binary Data

Undocumented BearShare-specific data used to sign QueryHits descriptors against fake hits coming from malicious servents. This field always comes after all extensions in the optional Private Data field, and is only present if the next field is also present, and its size is determined by the last field:

Optional Signature Size

BearShare servents using the authentication mechanism can add this field and the previous one to sign QueryHits descriptors so that fake QueryHits can be detected, and it may include other BearShare-specific information. This extension is still experimental. This field is a single unsigned byte, it indicates the size of the previous Signature Binary Data field. For compatibility, the signature field will always follow a set of GGEP extensions that must be terminated by a NUL GGEP extension. There should be only one set of GGEP extensions in the Private Data.

A.1.3. Vendor codes

These codes are used in several extensions of the Gnutella protocol. They were first introduced to be used in the BearShareTrailer EQHD, but they are now commonly used in other extensions as well.

Vendor Code	**Application Name** [4]
BEAR [5]	BearShare
CULT [5]	Cultiv8r (Emixode)
GNUC [5]	Gnucleus
GTKG [5]	Gtk-Gnutella
LIME [5]	LimeWire
RAZA [5]	Shareaza
SWAP [5]	Swapper
ACQX [6]	*Acquisition*
ARES [6]	*Ares (SoftGap)*
MACT [6]	*Mactella*
MMMM [6]	*Morpheus (v2.0+)*

MNAP [6]	*MyNapster*
MRPH [6]	*Morpheus (Old)*
MUTE [6]	*Mutella*
NAPS [6]	*NapShare*
PHEX [6]	*Phex*
QTEL [6]	*Qtella*
TGWC [6]	*???*
TOAD [6]	*ToadNode*
XOLO [6]	*Xolox*
FISH	*PEERanha*
GNOT	*Gnotella*
GNUT	*Gnut*
GNEW	*Gnewtellium*
HSLG	*Hagelslag*
OCFG	*OpenCola*
OPRA	*Opera*
SALM	*Salmonella*
SNUT	*SwapNut*
ZIGA	*Ziga*

[4] More information on these applications can be found at http://www.gnutelliums.com/.
[5] Servents that currently are very frequently found on the GNet, and using their own core engine.
[6] Servents that currently are commonly found on the GNet, and/or using a core engine from another vendor.
Vendors should register their peer code on the Database section of the following forum http://groups.yahoo.com/the_gdf/ where the protocol status and its revisions and extensions are discussed. This list only includes servents which registered with a released and active status.

A.2. Extended Result Data extensions

A.2.1. Extended Result structure

An extended Result Structure contains additional *Optional Result Data* between the two NUL terminators:

Fields	File Index	File Size	Shared File Name	NUL (0x0) Terminator	*Optional Result Data*	NUL (0x0) Terminator
Byte offset	0...3	4...7	8...7+K	8+K	*9+K...R-2*	R-1

Some servents may process extended *QueryHits* descriptors without adverse consequences by simply disregarding data between nuls. Others may, upon not finding an element of the result terminated as expected, improperly read the descriptor. Once misread, it may be subsequently mishandled, and the connection that delivered it may be disconnected.

The *Optional Result Data* field MAY NOT include any binary NUL byte. But for interoperability with other extensions, they may not include an ASCII FS (0x1c=28) character, which is used now to delimit multiple extensions in this field, and each extension should start with at least a magic byte characteristic of its content type.

This extension mechanism in this field using the FS separator is known as the Generic Extension Mechanism (GEM). Since the introduction of GGEP, no GEM extension in that field should start with a byte 0xC3 that delimits the start of a set of GGEP binary extensions, encoded with COBS (to avoid the presence of NUL bytes in the binary extension). IF GGEP extensions are present, no other GEM extension may follow it (the presence of a FS byte in the COBS-encoded GGEP set must not be detected as a new GEM extension).

A.2.2. Gnotella Result Data extension

Versions of the Gnotella client at least as early as 0.73 (released July 30, 2000) place extra data in *QueryHit* descriptors. According to the Gnutella 0.4 protocol specification, each element of the *Result Set* in a *QueryHit* descriptor is terminated by a double-nul. Gnotella may place extra data between the two nuls.

Although its exact layout is unknown, this data represents the bit-rate, sample rate, and playing time of the MP3 file described by the result set entry. If the file described by the result set entry is not an MP3 file, there is no data placed between the nuls.

This extension is now deprecated, as it lacks a properly defined structure with a recognizable content type.

A.2.3. LimeWire Meta-Data Result Data extension

Versions of LimeWire 1.8 or later can include meta-data for each Result inserted in a *QueryHits* descriptor. This meta-data consists in XML encoded items, and they are recognizable by their leading byte which is an ASCII "<" that starts an XML declaration, or XML comment, or the starting tag of the XML root element.

The root element MUST NOT be a simple text element, but an explicitly named element that encapsulates all the XML document, which must contain a element for each tagged set of meta-data, whose name qualifies the XML schema definition used to create

each meta-data field, encoded as indicated in the corresponding schema (for now, all meta-data information item is encoded as a separate sub-element, instead of attributes of the container element qualifying the schema).

This XML extension has no defined length, but consists only in printable ASCII bytes and bytes in the range 0x80 to 0xFE using the default UTF-8 encoding or another character encoding declared in a leading XML declaration; this extension is terminated on the first ASCII FS or NUL character that follows it.

Important interoperability notice: The XML standard (see section 2.8 of the XML 1.0 standard related to the "Prolog and Document Type Declaration", and section 4.3.3 for "Character Encoding in Entities") implies that the "**UTF-8**" charset is **implied by default** in all **conforming** XML documents, in absence of an explicit <?xml encoding="..."?> declaration at the beginning of the document (or without an leading "byte order mark" which may indicate "UTF-16", or "`ISO-10646-UCS-4`"). Also the default XML version is "1.0". Typical servents won't send this XML declaration.

So the "ISO-8859-1" character set MUST NOT be implied, but the 7-bit "US-ASCII" encoding can be used without any explicit declaration, as it is a common subset of both the "ISO-8859-1" and "UTF-8" encodings. Forgetting this conformance rule may simply invalidate the XML meta-data which won't be parsable by classic XML parsers such as Xerces for C or Java, or MSXML for Windows servents (so the XML meta-data will be ignored by the recipient). If an application does not wish to use the UTF-8 encoding, and prefer to use a legacy "ISO-8859-1" encoding for non "US-ASCII" characters, the XML meta-data document MUST start with one of the following explicit declarations:

```
<?xml encoding="ISO-8859-1"?>
<?xml version="1.0" encoding="ISO-8859-1"?>
```

Note that this XML extension may include 0xC3 bytes — needed for the UTF-8 encoding of Unicode characters in range U+00C0 to U+00FF, or if the ISO-8859-1 character set is declared, for encoding a "Ã" character (uppercase Latin letter A with tilde) — that MUST NOT be interpreted as prefixing a set of GGEP extensions.

Native encoding issues: Applications written in C/C++ (notably on Windows) MUST NOT assume that the system encoding is necessarily based on ISO-8859-1: extended characters in the range 0x80 to 0x9F are NOT part of ISO-8859-1 *(to use them, you'll need to encode your document with Unicode and serialize it in UTF-8, or specify an explicit "windows-1252" encoding)*, and the various native Windows "ANSI" codepages may be based on other ISO-8859 encodings *(notably ISO-8859-2 or ISO-8859-4 in Central Europe, ISO-8859-5 in Russia, ISO-8859-7 in Greece, ISO-8859-9 in Turkey, ISO-8859-8 in Israel, ISO-8859-6 in North*

Africa and Middle-East...), or on other Asian standards *(Shift-JIS in Japan, KSC5601 in Korea, GBK in China and Singapore, Big5 in Taiwan and Hong Kong, ISCII in India, TIS-620 in Thailand, VISCII in Vietnam)*. On Windows, applications can use the "GetACP()" Win32 API to get the "ANSI" codepage used on the system (for references about Windows codepages see http://www.microsoft.com/globaldev/reference/cphome.mspx). Applications written in Java internally use a Unicode representation for characters and strings independently of the supporting platform (so characters may be present outside of the ISO-8859-1 range).

Unicode representation issues: Unicode allows representing some characters in different ways, with a precombined form where a single codepoint represent a single character, or a decomposed form where multiple codepoints represent a single character as a base character and one or several combining characters. For interoperability of XML, the Unicode standard mandates using the canonical NFC "Normalized Form with Composed characters" in all XML documents. This may be an issue for interoperability, as some system APIs will return decomposed characters, but other will only operate with precomposed characters in NFC form (See http://www.unicode.org/reports/tr15/ for the Unicode Technical Report #15 about "Unicode Normalization Forms", and http://www.unicode.org/reports/tr20/ for the Unicode Technical Report #20 about "Unicode in XML and other Markup Languages").

UTF-8 serialization issues for Unicode: the UTF-8 encoding was initially used to serialize Unicode which used a single 16-bit code unit for each codepoint. However, the Unicode codepoints are now 21-bit entities (from a larger 32-bit "UCS4" set defined in ISO-10646), and some Unicode characters can only be represented in 16-bit code units using "surrogate" pairs, with a leading high surrogate in range 0xD800 to 0xDBFF, and a trailing surrogate in range 0xDC00 to 0xDFFF. These surrogates are NOT characters individually, so they MUST NOT be serialized to UTF-8 (using 3 encoding bytes for each surrogate). Instead, the surrogate pair must be converted as a whole, by first converting the pair to a 21 bit codepoint in range 0x10000 to 0x10FFFF, and then applying the UTF-8 serialization (which will return 4 encoding bytes, and not 6, for the whole character). This is important for compatibility with the new mandatory Chinese standard GB18030, which allocates characters out of the BMP. (See http://www.unicode.org/reports/tr26/ for details about the Unicode Technical Report #26 about "Compatibility Encoding Scheme for UTF-16: 8-Bit (CESU-8)", but don't use this "CESU-8" encoding representation which is not widely interoperable, and don't use this representation by assuming it is conforming to "UTF-8").

A.2.4. URI Result Data extension and URI Extended Query extension

Newer versions of servents that implement the "HUGE" extended protocol that enable identification of files by their content to provide accurate downloads from a mesh of servents, will include a URI for each Result inserted in a QueryHits descriptor.

An encoded URN are location independant and searchable on the network. An encoded URL will allow alternate download locations or download protocols.

URNs are recognizable by their leading byte, an ASCII "u" character that starts the URN encoding scheme. URLs MAY also be used, but they should only use well-known and registered schemes (such as "http:" or "ftp:").

Servents SHOULD only generate a URI in the Optional Result Data field of QueryHits Result structures only if requested to do so by using an URI Extended Query Data extension. In Query descriptors, no URI is inserted, only the URI encoding scheme is needed in the *Optional Query Data* extension field.

These URI extensions have no defined length, but consist only in printable ASCII bytes, and will terminate on the first ASCII FS or NUL character that follows each of them in QueryHits Result structures, and also by the end of the Payload of a Query Descriptor.

Home :: Developer :: Press :: Research :: Servents