Gnutella2 Specification Document

First Draft

2003-03-26

Table of Contents

/UPROD – User Profile Delivery

Introduction

Welcome

This is a specification and standard document for "Gnutella2". It is currently a "first public draft". Comments and revisions are welcome. This document covers the Gnutella2 architecture and standard only: it does not deal with Gnutella1 or other related standards, technologies and protocols that are well documented elsewhere.

What is Gnutella2?

Gnutella2 is a modern and efficient peer-to-peer network standard and architecture designed to provide a solid foundation for distributed global services such as person to person communication, data location and transfer and other future services.

Why is it needed?

Peer to peer technologies have become mainstream over recent years, and there are already a significant number of P2P networks in various stages of development and operation. How does yet another network help?

Gnutella2 is unique amongst the currently operating peer to peer networks in several important ways:

- Many of the most successful networks are "closed", owned by a single entity with restrictions or fees constituting a barrier to participation. This is not a viable model for an open, general purpose network. Gnutella2 is an open architecture where anyone is welcome to participate and contribute. The network has been designed to allow such diversity without the need for messy hacks or compromises in integrity.
- The majority of networks are devoted to a single purpose, often the sharing of files. This is certainly a popular application for peer to peer technology, but it is by no means the only application. Gnutella2 is designed as a general purpose network which can be used as a solid foundation for any number of different peer to peer applications – vanilla file sharing, communications tools or other ideas which are yet to be conceived.
- Some peer to peer networks have been developed with similar general purpose goals, however they have been unable to compete in the most popular application of the day, which is file sharing. For a general purpose network to succeed, it must be able to compete with purpose-specific networks in the most popular purpose. Gnutella2 is not only able to compete with the current popular file sharing specific networks, it outperforms them.

What About "Old Gnutella"?

The original "Gnutella" was created several years ago as a very simple, single vendor file-sharing specific network. Its simplicity made it a popular platform for file sharing application developers; however this simplicity also critically limited its effectiveness. As a result, competing file-sharing specific networks slowly but surely took over as the tools of choice as Gnutella users became frustrated with poor performance and turned elsewhere.

The original Gnutella[1] network was designed for a very limited purpose and, despite many changes over the years, remains limited today. Efforts to make it a better file sharing network continue with mixed success.

Gnutella2 shares the "Gnutella" name, striving to create the network that Gnutella should have been from the beginning. It shares the adopted ideals of openness and cooperation, but offers a fresh start that was sorely needed. The crippling limitations of the old network have been left behind and replaced with an entirely new network architecture ready to grow and develop through the creative efforts of many.

What is the Scope of Gnutella2?

The single name "Gnutella2" really refers to two separate components: Gnutella2 the Standard and Gnutella2

the Network.

The Gnutella2 Network is perhaps the most easily recognised component. It is a new high-performance peer to peer network architecture upon which a variety of distributed applications can be built, such as file sharing applications, communication tools, etc.

The Gnutella2 Standard is a set of requirements for building applications which operate on the Gnutella2 network in different capacities. It specifies the minimum compliance level required to be recognised as a Gnutella2-compatible application. Compliance with a Gnutella2 Standard ensures participating applications provide a minimum acceptable level of service to other network participants.

Gnutella2 the Standard

What is the Gnutella2 Standard?

The Gnutella2 Standard is a set of requirements for building applications which operate with the Gnutella2 network in different capacities. For example, the Gnutella2 Standard for File Sharing specifies a set of features and behaviours which must be available in any Gnutella2-connected file-sharing product offered to the public.

Why is a Standard Needed?

As an open, general purpose platform, Gnutella2 networks must be able to operate with a diverse family of different implementing applications. Every effort has been made to limit the ill-effects a non-compliant application can cause (deliberately or accidentally), however when it comes to critical features such as common URN schemes and character encodings, minimum standards help to ensure a favourable baseline user experience.

How are Standards Enforced?

The open and transparent nature of the Gnutella2 architecture makes technical enforcement difficult, so a more viable (and hopefully, more productive) social scheme has instead been adopted. Only applications meeting the appropriate Gnutella2 Standard may be marked as "Gnutella2-compliant". Websites containing information about Gnutella2 (such as gnutella2.com) are encouraged to list only compliant applications, and application developers are encouraged to deny communications with known non-compliant applications. Applications which do not comply with the standard or are still in the development process should never be made available to the public, however private testing is always encouraged.

How are Applications Tested?

Ultimately it is the responsibility of the developer to ensure their own application complies with the relevant standards, both with respect to Gnutella2 and any other functionality they may be including. However as an inter-dependent community, developers of Gnutella2-compliant applications are encouraged to take an interest in other Gnutella2 applications, and where possible, examine them for compliance. Similarly, new developers are strongly encouraged to seek assistance from other developers in verifying their work. This need not compromise competitive advantage – if the application is sensitive, the important compliance testing phase can be performed in the days prior to release.

What Standards are Available?

At the current time, only one Gnutella2 standard has been published: the Gnutella2 standard for File Sharing Applications, detailed below. Additional standards for other application classes will be published in the future as required.

Developers of new application classes are operating in somewhat untried territory, and should review the existing published standards for best practices which can be borrowed. In particular, the basic components of the Gnutella2 network architecture should always be implemented in full.

Common Gnutella2 Standard (All Applications)

All applications making use of Gnutella2 technology for any application class MUST IMPLEMENT the following

core features:

- Bidirectional TCP stream connections
(stream compression OPTIONAL)
- Bidirectional reliable UDP protocol
(Gnutella2 reliability layer and stateless compression REQUIRED)
- HTTP-style link negotiation, exchanging at least the required headers
- Gnutella2 protocol support, graceful handling of unknown trees
- Localised, UTF-8 and UNICODE decode REQUIRED, encoding to each optional
- Operation in LEAF mode, additional node states OPTIONAL
- Basic link handshaking and maintenance functionality (PI/PO/LNI/KHL)
- Global node addressing scheme and routing maintenance, addressing children (TO)
- Reverse (PUSH) connection response (connecting out)
- HTTP/1.1 client and server for peer to peer transactions

Gnutella2 Standard for File Sharing

Applications making use of Gnutella2 technology for file sharing MUST IMPLEMENT the following features:

- All of the COMMON features listed in the previous section
- Operation in LEAF mode, additional node states OPTIONAL
- Some form of bandwidth management scheme to keep network and transfer bandwidth below 95% of the user's link capacity – be it manually configured or some automatic scheme (very important to avoid flooding local connection)
- SHA1 and TIGER ROOT URNs for all shared objects
- XML metadata using existing schemas where appropriate (manual entry and peer acquired at minimum, automatic local collection highly recommended, service lookup optional)
- Universal 1-bit query hash filter, at least 2^20 length, intelligent density management scheme (superset combination required if supporting hub mode)
- Gnutella2 object search mechanism, all client responsibilities and if supporting hub mode, server responsibilities too
- Local search processing including simple query language (Boolean operations, quoted search terms, numeric range searches, interest flagging (I), local rule-based metadata searching)
- Extensible hit format (URN/DN/MD/URL are REQUIRED, all other extensions OPTIONAL)
- HTTP/1.1 based upload system, URN based requesting, partial content requests, active queuing, partial file uploading, timestamp protected alternate source cache and exchange
- Tiger Tree volume calculation on shared files, caching on downloads, exchange via DIME. Local corruption detection OPTIONAL but recommended.

Gnutella2 the Network Architecture

Introduction

The remainder of this document is concerned with the Gnutella2 network architecture, which is the most substantial part of the Gnutella2 initiative and forms the core of the Gnutella2 standard.

Components

The Gnutella2 network architecture consists of several key components:

- Node types and responsibilities for self-organising network topology
- TCP stream connection handshake negotiation and compression encoding
- UDP reliable/semi-reliable transceiver stack and encapsulation protocol
- Gnutella2 common tree packet structure (basic protocol)
- Basic network maintenance packet types
- Known hub cache and hub cluster cache
- Node route cache and addressed packet forwarding
- Query hash table, superset table and exchange packet types
- Gnutella2 object search mechanism, client and server roles, forwarding rules, filtering rules, security
- Local search responder with simple query language and metadata
- HTTP/1.1 server for upload queuing and servicing
- HTTP/1.1 client for download scheduling and transfer
- User profile challenge and delivery packet types

Note that a typical peer to peer application utilising Gnutella2 will have many additional components related to its core function. This list covers only components with an obvious network interaction, and should not be used as an exhaustive design guide. A typical file sharing application may have in excess of 120 components at the core level.

Node Types and Responsibilities

Introduction

The Gnutella2 network is an ad-hoc, self-organising collection of interconnected nodes cooperating to enable productive distributed activities.

Not all of the nodes participating in the system are equal: there are two primary node types, "hubs" and "leaves". The goal is to maximise the number of leaves and minimise the number of hubs, however due to the limited nature of resources the maximum viable ratio of leaves to hubs is limited. This quantity is known as the "leaf density".

Leaf Nodes

Leaf nodes are the most common node type on the network – they have no special responsibilities and do not form a working part of the network infrastructure. Nodes with limited resources must operate as leaf nodes: this includes limited bandwidth, CPU or RAM, low or unpredictable expected uptime, and inability to accept inbound TCP or UDP.

Hub Nodes

Hub nodes on the other hand form an important and active part of the network infrastructure, organising surrounding nodes, filtering and directing traffic over several media types. Hub nodes devote substantial resources to the network, and as a result their capacity to participate in higher level network functions is limited. Only the most capable nodes are selected to act as hubs, based upon the criteria in the following section.

Hub Selection Criteria

Hubs are selected based on the following internal criteria:

- Suitable operating system (able to support > 100 sockets)
- Suitably high CPU and RAM available
- Long uptime (many hours, at least two), possibly considering historical uptime

- Adequate bandwidth, primarily inbound bandwidth
- Ability to accept inbound TCP and UDP

In addition to these internal factors, hubs must also consider the network's need for additional hubs. Without a central point of authority the need for additional hubs cannot be determined with absolute certainty; however it can be approximated by examining the state of nearby nodes and specifically the state of hubs in the local hub cluster.

Hub Responsibilities

Nodes operating as Gnutella2 hubs have a set of responsibilities to meet. Hubs are highly interconnected, forming a "hub network" or "hub web", with each hub maintaining a connection to 5-30 other "neighbouring" hubs. The number of hub interconnections must scale up with the overall size of the network.

Each hub also accepts connections from a large collection of leaf nodes, typically 300-500 depending on available resources. Leaf nodes are considered to be the "edge" of the network. In practice leaves simultaneously connect to three hubs, however from the point of view of the hubs each leaf is considered a dead end.

The group of hubs within the hub network spanning the local hub and its neighbours is termed the "hub cluster", and is an important grouping. Hub clusters maintain constant communication with each other, sharing information about network load and statistics, exchanging cache entries and filtering tables. The hub cluster is also the smallest searchable unit of the network as far as a search client is concerned.

Hub responsibilities include:

- Maintaining up to date information about other hubs in the cluster, and their neighbouring hubs, providing updates to neighbours
- Maintaining a node routing table mapping node GUIDs to shortest route local TCP connections and UDP endpoints
- Maintaining query hash tables for each connection, including both leaves and hubs so that queries can be executed intelligently
- Maintaining a superset query hash table including local content and every connected leaf (not hub)'s supplied tables, to supply to neighbouring hubs
- Monitoring the status of local connections and deciding whether to downgrade to leaf mode, and keeping distributed discovery services such as GWebCaches updated

TCP Stream Connection and Handshaking

Introduction

TCP stream connections are established between Gnutella2 nodes when they elect to form a permeant link, creating the fundamental network topology of a highly interconnected hub network serving dense clusters of leaf nodes.

Initiation

TCP connections are initiated by leaf or hub nodes in an attempt to gain a connection to a hub node. Leaf nodes are never the target of an outbound connection. The TCP port number is not standardised, and must be stored with the IP address.

Handshaking

Upon the establishment of a TCP stream connection between two Gnutella2 nodes, a handshaking phase must be completed to negotiate the nature of the link and exchange other necessary information.

This handshaking phase is the only part of the communication which remains compatible with the old Gnutella network, allowing new connections to be negotiated without fore-knowledge the capabilities of the other node. The handshaking process has been well documented elsewhere, however a short summary is provided here.

Handshake Stages

The Gnutella handshake process consists of three header blocks. The node which initiated the connection sends an initial header block, of the form:

```
GNUTELLA CONNECT/0.6
Listen-IP: 1.2.3.4:6346
Remote-IP: 6.7.8.9
User-Agent: Shareaza 1.8.2.0
Accept: application/x-gnutella2
X-Ultrapeer: False
```

The receiver then responds with its own header block:

```
GNUTELLA/0.6 200 OK
Listen-IP: 6.7.8.9:6346
Remote-IP: 1.2.3.4
User-Agent: Shareaza 1.8.2.0
Content-Type: application/x-gnutella2
Accept: application/x-gnutella2
X-Ultrapeer: True
X-Ultrapeer-Needed: False
```

Finally, the initiator accepts the receiver's header block, and provides any final information:

```
GNUTELLA/0.6 200 OK
Content-Type: application/x-gnutella2
X-Ultrapeer: False
```

The latter two stages may be replaced with an error condition if the connection is being rejected. Appropriate error status codes are returned in this case, for example:

```
GNUTELLA/0.6 503 Too many connections
(more headers)
```

Note that only the HTTP-style error code should be interpreted by the software: any descriptive text provided is for display purposes only and is not standardised.

Important headers which are required or strongly recommended are detailed in the following sections.

Addressing Headers

Two important headers to send on all connections are "Remote-IP" and "Listen-IP". Both of these headers should be sent on the first transmission, meaning in the first and second header blocks in the three block exchange.

The **Remote-IP** header contains the IP address from which the *remote host* is connecting. This allows a remote host operating through some kind of network address translation system to learn its effective external address.

The **Local-IP** header contains the IP address and port number that the *local host* is listening for inbound TCP connections on. It should be listening for UDP datagrams on the same port. The format of this header is "IP:PORT", eg "1.2.3.4:6346".

Identification

The **User-Agent** header is used to identify the client software operating on the sending node. It should be sent on the first transmission, meaning in the first and second header blocks in the three block exchange. Note that this is a descriptive string that often includes a version number, and is not a "vendor code" as described elsewhere.

Content Type (Protocol)

The **Accept** and **Content-Type** header exchange is used to negotiate the data protocol that will be used in the connection, in this case Gnutella2. The Gnutella2 content type is "application/x-gnutella2", and this

exchange follows standard HTTP rules for negotiating content type.

The first step is to advertise support for the content type (Gnutella2) in the first header block with "Accept: application/x-gnutella2". The responding node will then indicate that it will send Gnutella2 content with "Content-Type: application/x-gnutella2", and that it also supports Gnutella2 with "Accept: application/x-gnutella2". The initiating host then confirms that it will be sending Gnutella2 with "Content-Type: application/x-gnutella2" in the third header block. For more information on the Accept/Content-Type exchange, consult a HTTP reference.

Note that the content type negotiation process is designed to be a "one way" process, i.e. a different content type can be negotiated for sending and receiving. However when the gnutella2 protocol is negotiated, both channels must use the same content type. This means that a receiving node must not accept Gnutella2 if the initiator did not advertise support for it, and if at the end of the handshake bidirectional Gnutella2 was not negotiated, the connection should be terminated.

Node State Negotiation

There are two node types in a Gnutella2 peer to peer network, a hub and a leaf as described in the "Node Types and Responsibilities" topic. During the initial handshake the two parties must exchange their current node type and advise of their capabilities, negotiating the node types they will adopt when the connection completes, and indeed whether it should complete at all.

As the handshake sequence is compatible with Gnutella1, the headers involved in negotiating node types are identical to those used to negotiate Gnutella1 "Ultrapeer" states:

```
X-Ultrapeer: [True|False]
X-Ultrapeer-Needed: [True|False]
```

Both headers contain a Boolean value, "true" or "false", case insensitive.

The **X-Ultrapeer** header indicates whether the transmitting node is currently operating as a hub. Hub nodes will send "X-Ultrapeer: True" while leaf nodes will send "X-Ultrapeer: False".

The **X-Ultrapeer-Needed** header indicates whether the transmitting node would like (and allow) the receiver to be a hub. A hub which sees no need for additional hubs in its area of the network will send "X-Ultrapeer-Needed: False", indicating to the connecting node that it must operate in leaf mode if it wishes to connect. A hub which sees a need for additional hubs will send "X-Ultrapeer-Needed: True", indicating that the receiving node should become a hub if it is capable of doing so. A leaf may send "X-Ultrapeer-Needed: True" to indicate that it is seeking a connection to a hub.

The **X-Ultrapeer** header should be sent on all three of the header blocks to indicate the current intended state of the node, while the **X-Ultrapeer-Needed** header should be sent in the first two header blocks only to indicate the desired status of the receiver. If the nodes cannot agree on a satisfactory arrangement, the connection will be terminated at or prior to the third header block with an appropriate message, for example:

```
GNUTELLA/0.6 503 Too many hub connections
GNUTELLA/0.6 503 Too many leaves
GNUTELLA/0.6 503 I have leaves, can't downgrade to leaf mode
GNUTELLA/0.6 503 Lead mode disabled
```

Hub Address Exchange

It is desirable for connecting nodes to exchange the node addresses of other hubs on the network to facilitate rapid connection. The Gnutella2 protocol includes highly efficient methods to share hub addresses with peers once connected, but for a node trying to connect and encountering only "full" hubs, learning new hubs to try is helpful.

The **X-Try-Ultrapeers** header was developed for this purpose, and like the rest of the handshake is also semi-compatible with Gnutella1. It contains a comma separated list of hub node addresses and ports, along with a timestamp recording the time the hub was last seen. For example:

```
X-Try-Ultrapeers: 1.2.3.4:6346 2003-03-25T23:59Z, [..more..]
```

Hub addresses should not be sent unless the transmitter has reasonable knowledge of the hub's existence and the timestamp is not too old. The content type negotiation headers should be used to verify that the listed hub addresses are indeed Gnutella2 hubs.

Compression Negotiation and Encoding

The Gnutella2 architecture makes widespread use of "deflate" compression, due to its high availability and ease of integration. Support of compressed TCP links is not a requirement in the Gnutella2 standard, however it is strongly recommended.

Deflate compression of a TCP link is negotiated with the standard HTTP headers pair "Accept-Encoding" and "Content-Encoding". For example:

Header block one (initiator):

```
Accept-Encoding: deflate
```

Header block two (receiver):

```
Accept-Encoding: deflate
Content-Encoding: deflate
```

Header block three (initiator):

```
Content-Encoding: deflate
```

In this example the initiator advertises *support* for a *receiving* a deflated connection. The receiver then indicates that it too supports *receiving* a deflated connection, and that it intends to transmit deflated data. Finally, the initiator upon noting that the receiver supports deflate indicates that it too will transmit deflated data.

Note that unlike the content-type / protocol negotiation, deflated encoding can be applied on either incoming, outgoing or both channels of a connection.

For performance reasons, nodes should consider whether they can afford to supports additional deflated connections before advertising support for them, or agreeing to provide a deflated data stream. In the Gnutella2 network topology, all links benefit from compression *except* the *leaf to hub* channel of the leaf/hub link. Exempting this channel from compression saves the leaf and more importantly the hub a considerable CPU and RAM investment.

Post Handshake Communication

After the third and final header block has been received by the initiator, subsequent communication over the TCP stream occurs in the negotiated protocol, with the negotiated encoding. This means that while the handshake sequence was backwards compatible with Gnutella1, after Gnutella2 support has been negotiated all subsequent communication occurs in the Gnutella2 common protocol – an entirely new system **not** backwards compatible any other protocol.

Gnutella2 UDP Semi-Reliable Transceiver

Introduction

The User Datagram Protocol (UDP) is invaluable in peer to peer systems because it provides a relatively low-cost (low-overhead) method of sending short, irregular messages to a very large number of peers on demand. Establishing a TCP stream connection to a peer simply to deliver a single packet of information is wasteful in data volume and time for the peers involved and state-aware network devices along the route, for example network address translation facilities. When dealing with a large number of peers quickly, these costs become unbearable. UDP provides a solution and makes this kind of interaction possible.

However the delivery of UDP packets is not reliable: packets may be lost en-route for a number of reasons. Often this behaviour is desirable, for example when the destination node's connection is highly congested, UDP packets are likely to be discarded. If the content was not critical, this loss is appropriate as the host's resources are effectively unavailable. In other scenarios involving critical payloads, UDP's lack of reliability is a problem: either the sender needs to make sure the receiver gets the payload, or it needs to know definitively that the receiver was unavailable.

The Gnutella2 network solves this problem by implementing a selectively engaged reliability layer on top of the basic UDP protocol. This reliability layer shares some common functionality with TCP, but importantly does not provide any connection state and thus retains the efficiency originally sought in UDP.

This allows Gnutella2 to select the most optimal communication medium for each and every transmission it needs to perform:

- If a significant volume of data is to be exchanged, or subsequent data will be exchanged with the same destination, a TCP connection is established
- If a small volume of important data is to be exchanged in a once-off operation or irregularly, reliable UDP is used
- If a small volume of unimportant data is to be exchanged in a once-off operation or irregularly, unreliable UDP is used

UDP

Gnutella2 semi-reliable communication is transported using the UDP protocol. The port number for receiving UDP is the same as the port number listening for TCP connections.

Encoding

Implementing an additional reliable protocol within UDP requires a small control header before the payload itself. This header is put to good use:

- A small signature identifies the packet as a Gnutella2 semi-reliable UDP datagram. This allows the same port to be used for receiving UDP traffic for other protocols if desired, and offers some simple protection against random, unexpected traffic.
- A content code identifies the payload as a Gnutella2 packet stream, allowing future protocols to be added within the same reliability layer if desired.
- Flags allow additional attributes to be specified, such as inline stateless compression of the payload (which is a required feature).

The header has a fixed size of 8 bytes, and is represented by the following C structure:

```
#pragma pack(1)
typedef struct
{
    CHAR  szTag[3];
    BYTE  nFlags;
    WORD  nSequence;
    BYTE  nPart;
    BYTE  nCount;
} GND_HEADER;
```

The members of the structure are detailed below:

- **szTag** contains a three byte encoding protocol identifier, in this case “GND” for “GNutella Datagram”. If this signature is not present the packet should not be decoded as a Gnutella2 reliability layer transmission.
- **nFlags** contains flags which modify the content of the packet. The low-order nibble is reserved for critical flags: if one of these bits is set but the decoding software does not understand the meaning, the packet must be discarded. The high-order nibble is reserved for non-critical flags: when set these bits may be interpreted, but an inability to interpret a bit does not cause the packet to be discarded. Currently defined flags are:

  0x01 : Deflate
  0x02 : Acknowledge Me

  When the deflate bit is set, the entire payload is compressed with the deflate algorithm. Note that the entire payload must be reassembled in the correct order before it can be deflated if the packet was fragmented. Fragments are not compressed separately!

  When the acknowledge me bit is set, the sender is expecting an acknowledgement for this packet.

- **nSequence** contains the sequence number of the packet. This sequence number is unique to the sending host only. It is not unique to the pair of the sending host and receiving host as in TCP, as there is no concept of connection state. Sequence numbers on consecutive packets need not be increasing (although that is convenient) – they must only be different. If a packet is fragmented, all of its fragments will have the same sequence number. Byte order is unimportant here.
- **nPart** contains the fragment part number (1 <= nPart <= nCount)
- **nCount** contains the number of fragment parts in this packet. On a transmission, this value will be non-zero (all packets must have at least one fragment). If nCount is zero, this is an acknowledgement (see below).

Fragmentation

Large packets must be fragmented before they can be sent through most network interfaces. Different network media have different MTUs, and it is difficult to predict what the lowest common size will be. Fragmentation and reassembly is performed by the existing Internet protocols, however there are two important reasons why the reliability layer performs its own fragmentation:

- Sockets implementations specify a maximum datagram size. This is adequate for the vast majority of transmissions, but it is desirable to have the transparent ability to send larger packets without worrying about the host implementation.
- When the Internet protocols fragment a packet and one or more fragments are lost, it may decide to discard the whole packet in an unreliable datagram protocol. The Gnutella2 reliability layer can compensate by retransmitting the whole packet, which would then be re-fragmented and each fragment resent – however this wastes the fragments that were successfully received before. Managing fragmentation natively allows this optimisation.

Each node determines its own MTU, often based on a best guess combined with information from the host's sockets implementation. Packets exceeding this size are fragmented into multiple datagrams of the appropriate size. Each datagram has the same sequence number and the same fragment count (nCount), but a different fragment number (nPart).

Transmission Process

When a packet is to be transmitted the network layer must:

- Cache the payload
- Allocate a new locally and temporally unique sequence number
- Derive the appropriate number of fragments
- Queue the fragments for dispatch
- If the fragments do not need to be acknowledged, the packet can be flushed now

The payload will generally be cached for an appropriate timeout period, or until the data cache becomes full at which time older payloads can be discarded. Fragments are dispatched according to the dispatch algorithm of choice, and the sender listens for acknowledgements. When an acknowledgement is received:

- Lookup the sent packet by sequence number
- Mark the nPart fragment as received and cancel any retransmissions of this part
- If all fragments have been acknowledged, flush this packet from the cache

If a fragment has been transmitted but has not been acknowledged within the timeout, it should be retransmitted. A finite number of retransmissions are allowed before the packet as a whole expires, at which time it is assumed that the packet was not received.

Receive Process

When a new datagram is received, the network layer must:

- If the acknowledge bit was set, send an acknowledge packet for this sequence number and part

number, with nCount set to zero (ack)

- Lookup any existing packet by the sending IP and sequence number
- If there is no existing packet, create a new packet entry with the IP, sequence number, fragment count and flags
- If there was an existing packet, make sure it is not marked as done – if so, abort
- Add the transmitted fragment to the (new or old) packet entry
- If the packet now has all fragments, mark it as done and decode it and pass it up to the application layer
- Leave the packet on the cache even if it was finished, in case any parts are retransmitted
- Expire old packets from the receive buffer after a timeout or if the buffer is full

Dispatch Algorithm

Fragment datagrams need to be dispatched intelligently to spread the load on network resources and maximise the chance that the receiver will get the message. To do this, the dispatch algorithm should take into account several points:

- Prioritize acknowledgements.
- If fragments are waiting to be sent to a number of hosts, do not send to the same host twice in a row. Alternating or looping through the target hosts achieves the same data rate locally, but spreads out the load over downstream links.
- Do not exceed or approach the capacity of the local network connection. If a host has a 128 kb/s outbound bandwidth, dispatching 32 KB of data in one second will likely cause massive packet loss, leading to a retransmission.
- After considering the above points, prefer fragments that were queued recently to older packets. A LIFO or stack type approach means that even if a transmitter is becoming backed up, some fragments will get there on time while others will be delayed. A FIFO approach would mean that a backed up host delivers every fragment late.

Parameters

The recommended parameters for the reliability layer are as follows:

MTU = 500 bytes

Transmit Retransmit Interval = 10 seconds

Transmit Packet Timeout / Expire = 26 seconds

(allows for two retransmissions before expiry)

Receive Packet Expiry = 30 seconds

(allows 10 seconds beyond final retransmission)

Performance Considerations

Relatively low-level network implementations such as this are reasonably complicated, but must operate fast. It is desirable to avoid runtime memory allocations in network code as much as possible, and particularly at this level.

It should be noted that in almost all cases, transmissions to "untested" nodes are single fragment. Replies on the other hand are often larger, and may be deflated in many fragments. This is optimal because attempting to contact a node which may be unavailable involves a retransmission of only a single fragment.

Flow control is an important topic, however it is handled at a higher layer. The UDP reliability layer is only responsible for guaranteeing delivery of selected datagrams.

Only critical transmissions whose reception cannot otherwise be inferred should have the acknowledge request

bit set.

Gnutella2 Packet Structure

Introduction

All Gnutella2 communications are represented with Gnutella2 lightweight tree packets. This applies everywhere from TCP stream communications to reliable UDP transmissions to HTTP packet exchanges (where protocol data has been negotiated). Each tree packet may contain meaningful payload data and/or one or more child packets, allowing complex document structures to be created and extended in a backward compatible manner.

The concept can be compared to an XML document tree. The "packets" are elements, which can in turn contain zero or more child elements (packets). The payload of a packet is like the attributes of an XML element. However serializing XML has a lot of overhead due to all the naming, even in a compact binary form. The Gnutella2 packet structure makes a compromise: it names elements (packets), allowing them to be globally recognized and understood without knowledge of their format – and stores attributes as binary payloads, requiring knowledge of their content to parse them.

Thus the element (packet or child packet) is the finite unit of comprehension. This system provides an excellent trade-off between format transparency and compactness.

Fictitious Visual Example

```
+ Query Hit Packet
|
|-+ Node ID (standard)
|
|-+ Server Status (standard)
| \-+ Shareaza Server Status (private extension)
|
|-+ Hit Object
| |-+ URN (standard)
| |-+ Descriptive name (standard)
| | \-+ Alternate name list (extension)
| |-+ URL (standard)
| |-+ Priority indicator (private extension)
| | \-+ Digital signature (private)
| |-+ Alternate source summary (standard)
| \-+ Available ranges (standard)
| . \-+ Estimated completion time (private extension)
|
|-+ Selective digital signature (private)
|
\-+ Routing tags
```

Contents

Each Gnutella2 packet contains:

- Control flags
- A type name meaningful in the namespace of the packet's parent or context
- A length (or implied length)
- Payload data of a format specific to the packet type name and namespace
- Child packets existing in the namespace of this packet

Namespace Considerations

Each packet contains a relative type name of up to 8 bytes in length, which are case sensitive. The packet

type name is meaningful only in the namespace of the packet's parent, or in the absence of a parent, the context of the packet (e.g. root level TCP stream).

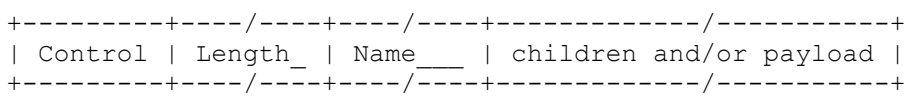
This means that, for example a packet "A" inside packet "X" is different to a packet "A" inside packet "Y". Packets are of the same type only if their fully qualified absolute type names are equal.

As a convention, when discussing packet type names, they will be noted in their absolute form with a URL style slash (/) separating each level. In the above example, the first packet is "/X/A" while the second is "/Y/A". It is clear now that the packets are of different types.

Packet type names can contain from 1 to 8 bytes inclusive, and none of these bytes may be a null (0). Community approved packets are by convention named with uppercase characters and digits, for example "PUSH". Private packet types are by convention named with lowercase characters and digits, prefixed with the vendor code of the owner, for example "RAZAclr2".

Framing

Packets are encoded with a single leading control byte, followed by one or more bytes of packet length, followed by one or more bytes of packet name/ID, followed by zero or more child packets (framed the same way), followed by zero or more bytes of payload:

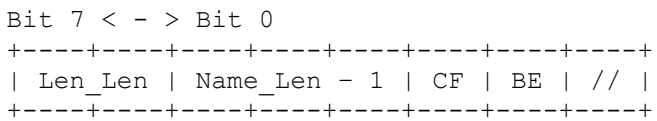


All packets can contain a payload only, children and a payload, children only, or nothing at all. The total length of the packet header (control, length and type name) cannot exceed 12 bytes and cannot be less than 2 bytes.

The Control Byte

The control byte is always non-zero. A zero control byte identifies the end of a stream of packets, and thus has special meaning. It should not be used in root packet streams (which do not end).

Control bytes have the following format:



- **Len_Len** is the number of bytes in the length field of the packet, which immediately follows the control byte. There are two bits here which means the length field can be up to 3 bytes long. Len_Len can be zero if the packet has zero length (no children and no payload), in which case there is no need to encode the length.
- **Name_Len** is the number of bytes in the packet name field MINUS ONE, which follows the packet length field. There are three bits here which means that packet names can be 1 to 8 bytes long inclusive. Because a 0 here equates to one byte of name, unnamed packets are not possible.
- The three least significant bits of the control byte are reserved for flags. They have the following meanings:
- **CF** is the compound packet flag. If this bit is set, the packet contains one or more child packets. If not set, the packet does not contain any child packets. If the packet is of zero length, this flag is ignored.
- **BE** is the big-endian packet flag. If set, all multi-byte values encoded in the packet and its children are encoded in big-endian byte order – including the length in the packet header.
- Other bits are reserved.

The Length Field

The length field immediately follows the control byte, and can be 0 to 3 bytes long. Length bytes are stored in the byte order of the packet.

The length value includes the payload of this packet AND any child packets in their entirety. This is obviously needed so that the entire packet can be detected and acquired from a stream. The length does not include the header (control byte, length, and name). The length field precedes the name field to allow it to be read faster from a stream when acquiring packets.

The length field is in the byte order of the root packet.

The Type Name Field

The type name field immediately follows the length bytes, and can be from 1 to 8 bytes long. Its format is detailed in the previous section entitled "Namespace Considerations".

Child Packets

Child packets are only present if the "compound packet bit" is set in the control byte. If set, there is one or more child packet immediately following the end of the header. These child packets are included in the total length of their parent (along with the payload, which follows the child packets after a packet stream terminator).

Child packets are framed exactly the same way, with a control byte, length, name, children and/or payload. When the compound bit is set and the packet is not of zero length, the first child packet must exist. Subsequent child packets may also exist, and are read in sequentially in the same way that they are read from a root packet stream. The end of the child packet stream is signalled by the presence of a zero control byte, OR the end of the parent packet's length (in which case there is no payload). Including a terminating zero control byte when there is no payload is still valid, but unnecessary.

Payload

Payload may exist whenever the length field is non-zero. However, if the compound bit is set, one or more child packets must be read before the payload is reached. If there is no packet left after the end of the last child, there is no payload.

Notes on the Control Byte

Note that there are a number of "marker packet types", which have no children or payload. It is desirable to encode these in as small a space as possible, which means omitting the length field and setting the len_len bits to zero in the control byte. This creates a potential conflict, as the control byte itself may be zero if the type name is one byte long – and as noted above, a zero control byte has special meaning (end of packet stream). This must be avoided; luckily it is perfectly legal to set the compound packet flag (CF) on zero length packets, thus producing a non-zero control byte and the most compact packet possible.

The compound packet bit MUST be checked when decoding every packet. It should be done in low-level decoding code to avoid accidental omission. Do not assume that a packet will not have children – it might not now, but no packets are sterile. Anything could be augmented or extended in some unknown way in the future. If you are not interested in children, skip them (which is easy, you don't even need to recurs through their children).

Simple Packet Decoder in C

```
BYTE nInput           = ReadNextByte();
if ( nInput == 0 ) return S_NO_MORE_CHILDREN;

BYTE nLenLen    = ( nInput & 0xC0 ) >> 6;
BYTE nTypeLen   = ( nInput & 0x38 ) >> 3;
BYTE nFlags           = ( nInput & 0x07 );
```

```
BOOL bBigEndian       = ( nFlags & 0x02 ) ? TRUE : FALSE;
BOOL bIsCompound      = ( nFlags & 0x04 ) ? TRUE : FALSE;

ASSERT( ! bBigEndian );

DWORD nPacketLength = 0;
ReadBytes( (BYTE*)&nPacketLength, nLenLen );

CHAR szType[9];
ReadBytes( (BYTE*)szType, nTypeLen + 1 );
szType[ nTypeLen + 1 ] = 0;
```

Datatypes

Introduction

The format of a packet payload is defined by the packet type and can consist of any binary data; however there are a number of conventions in place for serializing common datatypes.

Multi-Byte Integers

Multi-byte integers are serialized in the byte-order of the topmost packet. Little-endian is the default byte-order; however big-endian byte order can be selected for those who want it.

Network/Node Addresses

A network or node address consists of a physical address and a port number, and are of variable length depending on the address family.

In IPv4, a network/node address is six bytes long: 4 bytes for an IP address and 2 bytes for a port number as follows:

```
typedef struct
{
    BYTE  ip[4];
    SHORT port;
} IPV4_ENDPOINT;
```

Note that this is considered an array of 4 8-bit integers (bytes), followed by a 16-bit integer (short). Byte order does not affect bytes, but it will affect the 16-bit port number.

IPv6 addresses are longer and are not yet defined within the scope of Gnutella2, however applications should be aware that if the node address is not 6 bytes it is of a different address family.

GUIDs

Globally unique identifiers (GUIDs) are used to identify nodes on the network. GUIDs are serialized as an array of 16 bytes.

Strings

Strings are serialized as a zero-terminated array of 8 or 16 bit integers, with byte order considerations applying if 16 bit integers are used. 16 bit storage is selected by storing a single 0xFF byte before the string – a traditional byte-order mark is not required because the byte order has already defined for the packet as a whole.

A zero character (0x00 or 0x0000) marks the end of the string, however if the string data meets the end of the packet (or child packet) payload, the terminator is not required. This means that packets whose payload consists of a string do not need to include a zero string terminator and their payload length will be the byte length of the encoded string exactly.

When 16-bit mode is selected, the string is serialized in 16-bit Unicode. Most runtime environments provide tools for working with 16-bit Unicode text.

When 8-bit mode is selected, the string is serialized in UTF-8. 7-bit characters are passed as-is, while extended characters are encoded with multi-byte sequences.

All applications are required to support decoding both 8 and 16 bit strings, however it is up to each application which string format to encode with. For example it is acceptable to send all strings as 16-bit Unicode if desired.

Basic Network Maintenance

Introduction

Once TCP stream connections have been established forming the overall network topology, a set of basic packet types and behaviours are used to keep the network together and perform maintenance functions.

These maintenance packets are documented in this section, each with name, payload, children and behaviours.

/PI - (Ping)

Overview

The ping packet is used to verify the presence of the addressed node, testing that:

- The destination node is online
- The destination node can receive the transmission
- The local node can receive the reply

Aside from performing a "keep-alive" function on TCP links, pings can be used to solicit inbound UDP datagrams to test the local node's ability to receive UDP. Many NAT systems will route UDP traffic from nodes with whom a persistent TCP stream has been established, so it is necessary to test inbound traffic from nodes with which there is no TCP connection. This is achieved by sending a ping with a /PI/UDP packet to a connected hub via TCP, which will forward the request to its neighbours who then reply to the originator via UDP.

Sending

On a TCP stream connection, if one end has not received a valid packet from the other after an internally determined period of time, it has the option of sending a keep-alive ping to verify the other end's presence. If a pong is not received in a timeout period it may close the connection.

If a node needs to verify that it is able to receive UDP datagrams from nodes with which it does not have a TCP stream connection, it may send a ping packet with a /PI/UDP child packet. If it then receives a /PO (pong) packet with a /PO/RELAY child tag and there is no TCP connection to the sender, it can assume it is able to receive UDP.

Receiving

Upon receiving a keep-alive ping, a node should respond with a keep-alive pong immediately.

If the ping was received from a TCP neighbour and contains a /PI/UDP child packet, but no /PI/RELAY tag, a /PI/RELAY tag should be added and the packet forwarded to all neighbouring hubs. If a /PI/RELAY tag was present, the node should send a /PO packet with a /PO/RELAY tag to the UDP address identified in the /PI/UDP child packet.

Payload

This packet has no payload at the current time.

Children

The ping packet has two child packets defined at the current time:

- /PI/RELAY
- /PI/UDP

/PI/RELAY – Relayed Ping Marker

Overview

If a /PI packet contains a /PI/RELAY child marker, the ping has been relayed for a third party. The receiving node should reply to the original party rather than the sender. The original party should be contacted via UDP with the address contained in the /PI/UDP child. If the /PI/UDP child is absent, the packet is invalid.

Sending

The /PI/RELAY child should be added to a /PI packet when relaying a ping to hub neighbours at the request of a leaf.

Receiving

Upon receiving a /PI packet with a /PI/RELAY tag, a /PO (pong) packet should be sent via UDP to the node identified in the /PI/UDP child.

Payload

This packet has no payload at the current time.

Children

This packet has no known children at the current time.

/PI/UDP – Ping Response Address

Overview

The /PI/UDP child packet specifies the return address for a relay able or relayed ping.

Sending

Add a /PI/UDP child packet to a /PI packet to request a relayed "two hop" ping.

Receiving

The /PI/UDP child packet indicates that a /PI packet needs to be relayed.

Payload

The /PI/UDP packet contains an endpoint address.

Children

This packet has no known children at the current time.

/PO – Pong

Overview

The pong packet serves as a reply to a one or two hop ping (/PI) packet. It indicates that the pinged node received and processed the ping, and in the case of UDP, that the local node was able to received UDP from the sender.

Sending

Send a pong packet in response to a direct or relayed ping request.

Receiving

No action is required in response to a pong packet. If the packet contains a /PO/RELAY marker child, test the sender's network address against the list of active TCP connections and if there is no match, consider the local node able to receive UDP.

Payload

This packet has no payload at the current time.

Children

The ping packet has one child packet type defined at the current time:

- /PO/RELAY

/PO/RELAY – Relayed Pong Marker

Overview

If a /PO packet contains a /PO/RELAY child marker, the pong was sent in response to a relayed ping.

Sending

The /PO/RELAY child marker should be added to a /PO if the pong is being sent in response to a relayed ping.

Receiving

No specific action is required for a relayed pong.

Payload

This packet has no payload at the current time.

Children

This packet has no known children at the current time.

/LNI – Local Node Information

Overview

The local node information packet is used to convey essential information about the node on either end of a TCP stream connection.

Sending

The /LNI packet should be sent to a TCP neighbour upon connection, and at regular intervals following that if the information within it has changed. A minimum update time of one minute is recommended.

Receiving

Upon receiving a /LNI packet, the information within should be stored for subsequent use.

Payload

This packet has no payload at the current time.

Children

The /LNI packet has many child packet types defined at the current time:

- /LNI/NA – Node Address

- /LNI/GU – GUID
- /LNI/V – Vendor Code
- /LNI/LS – Library Statistics
- /LNI/HS – Hub Status

/LNI/NA – Node Address

Overview

The /LNI/NA child packet specifies the node or network address of the sending node.

Sending

This child is required.

Payload

The physical network address of the sending node. See datatypes for more information.

Children

This packet has no known children at the current time.

/LNI/GU – GUID

Overview

The /LNI/GU child packet specifies the globally unique identifier of the sending node.

Sending

This child is required.

Payload

The 16 byte globally unique node identifier of the sending node.

Children

This packet has no known children at the current time.

/LNI/V – Vendor Code

Overview

The /LNI/V child packet specifies the vendor code of the software operating the sending node.

Sending

This child is optional.

Payload

A four byte vendor code.

Children

This packet has no known children at the current time.

/LNI/LS – Library Statistics

Overview

The /LNI/LS packet provides information about the content library of the sending node.

Sending

This child is optional.

Payload

Two 32-bit integers representing the number of files in the local library, and the total KB in the local library respectively. If the sending node is a hub these statistics are combined with the known statistics from all connected leaves. The payload may grow beyond 8 bytes in the future.

Children

This packet has no known children at the current time.

/LNI/HS – Hub Status

Overview

The /LNI/HS packet is included only if the sending node is a hub, and contains the status of the hub.

Sending

This child packet should only be sent by hub nodes. Leaf nodes should not transmit it, and if it is received from a leaf node, it should be ignored.

Payload

Two 16-bit integers representing the current leaf count and the maximum leaf count respectively. This packet may grow beyond 4 bytes in the future.

Children

This packet has no known children at the current time.

/KHL – Known Hub List

Overview

The known hub list packet is used to exchange a list of known Gnutella2 hubs with neighbouring nodes. It is exchanged regularly over all TCP connections, between hubs and between hubs and leaves (and vice versa). It enables a completely up to date picture of the local hub cluster, and the gradual propagation of cached hub records.

Sending

This packet should be sent through TCP links at regular intervals. One minute is suggested for busy links, leaves may send to hubs less often. Note that unlike /LNI, /KHL must be sent regularly even if the information does not change in order to refresh timestamps and keep time-sorted caches in the correct order.

Receiving

Upon receiving a known hub list, the detailed hubs should be stored in appropriate data structures. Neighbouring hubs must be retained in the hub cluster records, while cached hubs should be merged with the local accessible hub cache.

Payload

This packet has no payload at the current time.

Children

This packet has several child packet types defined at the current time:

- /KHL/TS - Timestamp
- /KHL/NH – Neighbouring Hub
- /KHL/CH – Cached Hub

/KHL/TS – Timestamp

Overview

The /KHL/TS child provides a timestamp representing the current universal time at the sending node. This can be used as a reference when considering other timestamps in the packet, allowing them to be adjusted to eliminate differences between the time setting on the local and remote node.

Sending

This child is optional but recommended.

Payload

A 32-bit integer representing the current UNIX time, or time (NULL).

Children

This packet has no known children at the current time.

/KHL/NH – Neighbouring Hub

Overview

This child packet represents a neighbouring hub.

Sending

One should be included for each Gnutella2 neighbour hub, with the possible exception of the one to which this packet is being sent.

Payload

The node address of the hub. See datatypes for more information.

Children

This packet has several child packet types defined at the current time:

- /KHL/NH/GU - GUID
- /KHL/NH/V – Vendor Code
- /KHL/NH/LS – Library Statistics
- /KHL/NH/HS – Hub Status

/KHL/NH/GU – GUID

Overview

The /KHL/NH/GU child packet specifies the globally unique identifier of the hub.

Sending

This child is optional.

Payload

The 16 byte globally unique node identifier of the hub.

Children

This packet has no known children at the current time.

/KHL/NH/V – Vendor Code

Overview

The /KHL/NH/V child packet specifies the vendor code of the software operating the hub.

Sending

This child is optional.

Payload

A four byte vendor code.

Children

This packet has no known children at the current time.

/KHL/NH/LS – Library Statistics

Overview

The /KHL/NH/LS packet provides information about the content library of the hub and its connected leaves.

Sending

This child is optional.

Payload

Two 32-bit integers representing the number of files, and the total KB of available content in the hub's own library and the libraries of its connected leaves. The payload may grow beyond 8 bytes in the future.

Children

This packet has no known children at the current time.

/KHL/NH/HS – Hub Status

Overview

The /KHL/NH/HS packet contains the status of the hub.

Sending

This child is optional.

Payload

Two 16-bit integers representing the current leaf count and the maximum leaf count respectively. This packet may grow beyond 4 bytes in the future.

Children

This packet has no known children at the current time.

/KHL/CH – Cached Hub

Overview

This child packet represents a hub from the hub cache.

Sending

An arbitrary number of these children should be included, constrained by space and the number of available cached hubs. A selection of freshly time stamped but previously unknown hubs should be included, and the list should not include hubs in the local cluster.

Payload

The node address of the hub, followed by the time this hub was last seen as a 32-bit timestamp. The last seen time can be adjusted based on the difference between the /KHL/TS timestamp and the local time. Note that the node address is of variable length, so it is important to consider the last 4 bytes as the timestamp rather than assuming a fixed length node address.

Children

This packet has several child packet types defined at the current time:

- /KHL/CH/GU - GUID
- /KHL/CH/V – Vendor Code
- /KHL/CH/LS – Library Statistics
- /KHL/CH/HS – Hub Status

This extra information is considered less relevant for a cached hub because it is unlikely to be up to date enough to be useful.

/KHL/CH/GU – GUID

Overview

The /KHL/CH/GU child packet specifies the globally unique identifier of the hub.

Sending

This child is optional.

Payload

The 16 byte globally unique node identifier of the hub.

Children

This packet has no known children at the current time.

/KHL/CH/V – Vendor Code

Overview

The /KHL/CH/V child packet specifies the vendor code of the software operating the hub.

Sending

This child is optional.

Payload

A four byte vendor code.

Children

This packet has no known children at the current time.

/KHL/CH/LS – Library Statistics

Overview

The /KHL/NH/LS packet provides information about the content library of the hub and its connected leaves.

Sending

This child is optional.

Payload

Two 32-bit integers representing the number of files, and the total KB of available content in the hub's own library and the libraries of its connected leaves. The payload may grow beyond 8 bytes in the future.

Children

This packet has no known children at the current time.

/KHL/CH/HS – Hub Status

Overview

The /KHL/CH/HS packet contains the status of the hub.

Sending

This child is optional.

Payload

Two 16-bit integers representing the current leaf count and the maximum leaf count respectively. This packet may grow beyond 4 bytes in the future.

Children

This packet has no known children at the current time.

Known Hub Cache and Hub Cluster Cache

Introduction

Each Gnutella2 node must maintain a non-exhaustive cache of known hubs at the global level, and an exhaustive cache of hubs within the neighbouring hub cluster(s).

The Known Hub Cache

The known hub list is used to provide connection hints to the local connection manager, and other nodes which connect permanently or transiently. The most recent portion of it is exchanged with neighbours regularly.

It is also used when executing a query on the network, which involves iteratively contacting hubs representing each hub cluster and simultaneously recording the hubs which have been accounted for and adding those newly discovered.

The known hub cache should be highly efficient, addressable by node address and timestamp, and sorted by the last seen timestamp of each hub record. Adding fresh hubs should push the oldest hubs from the bottom

of the cache.

It is suggested that each cached hub entry store:

- Node address
- Last seen time
- Query key, if available
- GUID, if available
- Vendor information, if desired and available
- Throttling timing for last query key request, last query, etc

Hubs whose last seen timestamp is too old should be removed from the cache, and should certainly never be sent to other nodes.

The Hub Cluster Cache

The hub cluster cache is used to maintain an up to date list of neighbouring hubs, and the neighbouring hubs of neighbouring hubs. Thinking in terms of hops, it is an exhaustive list of every hub which is 1 or 2 hops away from the local node (be the local node a hub or a leaf). The cluster cache is really only important when operating in hub mode, however it can be maintained in leaf mode for informational purposes.

The hub cluster cache is used by a hub when responding to a keyed remote query request, in the generation of a query acknowledgement packet (/QA). The /QA packet contains a list of neighbouring hubs and a selection of second degree neighbours.

The cluster cache is updated using information from /LNI and /KHL packets.

Node Route Cache and Addressed Packet Forwarding

Introduction

Each Gnutella2 node must maintain a dynamic node route cache to map node GUIDs to appropriate destinations. The route cache is consulted when a packet needs to be dispatched to or toward a GUID-addressed node. GUID-addressing is used over network-addressing in a number of situations.

Data

Each entry in a node route cache will have:

- The GUID of the target node
- The local TCP connection providing the best route to the node, or
- A UDP endpoint for the node or the best route to the node

Performance

A route cache needs to be addressable by GUID, and must implement a refresh mechanism to store routes for an appropriate amount of time based upon route hits. Many schemes exist to engineer efficient lookup systems, such as hash tables, two-table exchanges and balanced trees.

The route cache is similar to the several GUID mapping caches involved in old Gnutella applications, with two key differences:

- Each entry may map to a local address **or** a UDP endpoint
- There is only one unified route cache for all purposes

Applications

GUIDs are used to identify virtual entities within the network, such as nodes (node GUIDs), searches (search GUIDs) and other transactional requests. The GUIDs associated with these entities can then be committed to

the route cache and mapped to reflect the easiest route back to the owner of the object.

All nodes should be aware of their directly connected neighbours, and treat these node GUIDs as a special case that need never expire.

Addressed Packet Forwarding

Any Gnutella2 packet may be addressed to a particular destination node by GUID. Upon receiving an addressed packet, it should be immediately forwarded either to the destination, or toward it. Addressed packets should not be interpreted locally unless the destination address matches the local GUID.

Loops are avoided by placing restrictions upon forwarding:

- If received from a leaf via TCP, a packet may be forwarded anywhere
- If received from a hub, a packet may only be forwarded to a leaf
- If received via UDP, a packet may not be forwarded via UDP

Note that these restrictions apply only to generically addressed packets. Some packet types have specific forwarding rules which override these. These rules allow any node to be reached in two hops.

Packets are addressed by including a special child packet as the first child of the root packet. The child packet is named "TO", so its full name would be "/?/TO" where ? is the root packet name. The address packet's payload consists of a 16 byte GUID, and it has no children defined at the current time.

Reverse Connection (Push) Requests

Addressed packet forwarding can be used to deliver any valid Gnutella2 communications to leaf nodes that are unable to accept TCP or UDP traffic directly. The packet is simply sent to one of the hubs to which the target leaf is attached, or a hub in the same hub cluster, and the forwarding rules will allow the packet to reach its destination.

This mechanism is used to request that a "firewalled" leaf initiate a "reverse connection" or "call-back" to an elected address. The root packet type "/PUSH" is used, and is documented in the following sections.

/PUSH – Push / Connect Back Request

Overview

The push packet is used to request that a node initiate a TCP connection to a specified node address. It is useful where the receiving node is unable to accept TCP connections naturally.

Sending

An addressed /PUSH packet should be sent to a hub with direct or proxy access to the destination leaf node. It can be sent directly by a hub to one of its leaves without the need for addressing.

Receiving

Upon receiving a /PUSH packet, the local node should initiate a TCP connection to the elected node address and issue a "push handshake", documented below.

Payload

The network address of the node to contact, including of course the port number.

Children

This packet has no known children at the current time, however it is often used with the generic addressing child packet "/?/TO" to indicate that it should be forwarded to a destination.

Push Handshaking

The Gnutella2 push handshake is very simple – it simply identifies the connection as originating from a push,

and provides the GUID of the initiating node:

```
PUSH guid:GUIDinHEXguidINhexGUIDinHEXguidI     (\r\n)
(\r\n)
```

Note that unlike the Gnutella1 case, no purpose-specific information is provided. The pushed connection can now be used for any purpose a normally established TCP link could adopt, including but not limited to:

- Gnutella2 network connection
- Data transfer
- Personal communications
- Etc

Gnutella2 implementations are however strongly advised to provide backward support for the Gnutella1 push handshake, even if not supporting Gnutella1 directly. This is because applications supporting both protocols may be unable to determine in advance whether the host they are pushing to is Gnutella1 or Gnutella2.

The legacy-style push handshake looks like:

```
GIV 0:GUIDinHEXguidINhexGUIDinHEXguidI/ (\n\n)
```

The leading zero and trailing slash have task-specific meanings in Gnutella1; however Gnutella2 applications can safely ignore them and consider only the GUID. Be sure to allow for variable length handshakes, however.

Query Hash Tables, Superset Table and Exchange

Introduction

Building an efficient network topology is not viable without means to restrict the flow of information appropriately. In the Gnutella2 network architecture, neighbouring nodes exchange compressed hash filter tables describing the content which can be reached by communicating with them. These tables are updated dynamically as necessary.

The concept of filtering hash tables in file sharing was pioneered by Limegroup for the LimeWire Gnutella1 application.

Table Properties

Query hash tables or QHTs provide enough information to know with certainty that a particular node (and possibly its descendants) will **not** be able to provide any matching objects for a given query. Conversely the QHT may reveal that a node or its descendants **may** be able to provide matching objects.

This property means that queries can be discarded confidently when a transmission is known to be unnecessary, while not providing the filtering or forwarding node any actual information about the searchable content. Neighbours know what their neighbours do not have, but cannot say for sure what they do have. QHTs are also very efficient both in terms of exchange and maintenance and lookup cost.

Table Content

A QHT is a table of 2^N bits, where each bit represents a unique word-hash value. For example a table of 20 bits has 2^20 = 1048576 possible word-hash values. If stored uncompressed this table would be 128 KB in size.

In an empty table, every word-hash value bit will be "1", which represents "empty". To populate the table with searchable content, an application must:

- Locate every plain-text word and URN which could be searched for and produce a match/hit
- Hash the word with a simple hash function to produce a word-hash value which is 0 <= value < (2^N).

- Set the appropriate bit in the table to zero, representing “full”.

Word Hashing

Words are strings of one or more alphanumeric characters which are not all numeric.

To convert a word into a hash value, the following case-insensitive algorithm is used:

```
// HashWord( string_ptr, char_count, table_bit_count );

DWORD CQueryHashTable::HashWord(LPCTSTR psz, int nLength, int nBits)
{
    DWORD nNumber     = 0;
    int nByte         = 0;

    for ( ; nLength > 0 ; nLength--, psz++ )
    {
          int nValue = tolower( *psz ) & 0xFF;

          nValue = nValue << ( nByte * 8 );
          nByte = ( nByte + 1 ) & 3;

          nNumber = nNumber ^ nValue;
    }

    return HashNumber( nNumber, nBits );
}

DWORD CQueryHashTable::HashNumber(DWORD nNumber, int nBits)
{
    WORD64 nProduct = (WORD64)nNumber * (WORD64)0x4F1BBCDC;
    WORD64 nHash = ( nProduct << 32 ) >> ( 32 + ( 32 - nBits ) );
    return (DWORD)nHash;
}
```

URNs – A Special Case

URNs are treated as a special case: rather than dividing them up into word tokens, they are hashed as a complete fixed length string. For example:

```
urn:sha1:WIXYJFVJMIWNMUWPRPBGUTODIV52RMJA
```

Bitprint URNs are actually composite values which include both a SHA1 and TigerTree root value. Rather than adding the whole bitprint to the table, each of the constituent URNs are added separately. This allows SHA1-only querying and tiger-only querying. A root TigerTree URN takes the form:

```
urn:tree:tiger/:CN25MLNU3XNN7IHKZMNOA63XG6SKDJ2W7Z3HONA
```

Other URNs should be expressed in their most natural form before being fed to the word hash function.

Word Prefix Extensions

For words consisting of at least five characters, it is often useful to be able to match substrings within the word. Unfortunately adding every possible substring of each word would increase the density of the QHT, however a simple and effective compromise is available:

For words with 5 or more characters:

- Hash and add the whole word
- Hash and add the whole world minus the last character
- Hash and add the whole world minus the last two characters

This allows searching on prefixes of the word, for example “match” will now match “matches”.

Table Exchange

Nodes must keep their neighbouring hubs up to date with their latest local query hash table at all times. Rather than sending the whole table whenever it changes, nodes may opt to send a “table patch”, which includes only the difference between the old and new table.

The /QHT packet is used to supply a query hash table update to a neighbouring hub. Its format is compatible with the Gnutella1 “query routing protocol”, except that Gnutella2 requires a 1-bit per value table while Gnutella1 requires a 4 or 8 bit per value table. Gnutella2 supports patch compression using the deflate algorithm, however this should not be applied if the TCP link itself is compressed.

/QHT – Query Hash Table

Overview

The query hash table update packet supplies an update to the query hash table for the sending node. It supports two operations: clear existing table, and patch existing table with new data.

Sending

The /QHT packet should be sent when the local or aggregate query hash table has changed.

Receiving

Upon receiving a /QHT packet, the local hub should patch the query hash table for the connection on which the packet was received. If the connection is to a leaf node, the hub should update its aggregate table and queue a propagation of that table to neighbouring hubs in the cluster.

Payload

The payload is of the same format as Gnutella1’s “query routing protocol” message:

The first byte is a command byte, 0x00 indicates a reset command, while 0x01 indicates a patch command.

Payload – Reset

```
typedef struct
{
    BYTE  command;                // = 0 (reset)
        LONG  table_size_in_entries;  // = 2^(size_in_bits)
    BYTE  infinity;               // = 1
} QHT_RESET;
```

A reset packet establishes a new table and clears it (replacing any existing table). In Gnutella2 the value of infinity must always be 1. Note that the table size is measured in entries rather than bits, so it is equal to 2^N where N is the number of bits.

Payload – Patch

```
typedef struct
{
    BYTE  command;          // = 1 (patch)
    BYTE  fragment_no;      // = number of this patch fragment
    BYTE  fragment_count;   // = number of patch fragments
    BYTE  compression;      // = 0 for none, = 1 for deflate
    BYTE  bits;             // = 1 for Gnutella2
    (data)
    } QHT_PATCH;
```

Patches may be fragmented into more than one packet if necessary, in which case the fragment number and

count fields are used to reassemble the patch. Note that fragments must be sent in correct order.

Patches may be compressed – in this case the whole patch is compressed, rather than compressing fragments individually. One compression type is defined, deflate compression.

Gnutella2 always uses tables with 1 bit per entry, so the bit count must be 1.

The patch data in uncompressed form is of (2^N)/8 bytes length, where N is the table bit size. This is exactly the same size as the table which is being patched. To apply the patch, simply XOR the existing table with the patch data. In other words, whenever there is a 1 bit, the bit in the existing table is toggled.

Table Access

A table of 2^N bits can be stored in an array of bytes (2^N)/8 long. To resolve a hash-value into a byte and bit number, use the following equations:

```
int nByte = ( nHashValue >> 3 );
int nBit  = ( nHashValue & 7 );
```

The least significant bit is numbered 0; the most significant bit is numbered 7. To set a bit as empty (setting it to 1):

```
table_ptr[ nByte ] |= ( 1 << nBit );
```

To set a bit as full (setting it to zero):

```
table_ptr[ nByte ] &= ~( 1 << nBit );
```

The Aggregate or Superset Table

Nodes operating in hub mode must maintain an aggregate or superset query hash table, consisting of their own searchable content combined with the QHTs supplied by all connected leaves. This aggregate table is supplied to neighbouring hubs, allowing them to completely filter traffic for the local hub and its leaves.

This has two important implications:

- When a change is detected in either the local content or a connected leaf node's QHT, the aggregate table must be rebuilt and patches dispatched to neighbouring hubs. This will happen often, so an appropriate minimum time between updates should be used. One minute is effective.
- An aggregate table representing 400 leaves will be much denser than a table representing one node. This means that **all tables** must be large enough that the aggregate table remains productively sparse.

To create an aggregate table, start with an empty table of fixed size containing all 1's. For each contributing table, copy any zero (full) bits to the aggregate table. This is effectively an AND operation. If a source table is smaller than the aggregate table, a single 0 bit in the source will equate to several zero bits in the aggregate. If the source table is larger than the aggregate table, a single zero bit will map to a single bit with some loss of accuracy.

It is of great importance that all QHTs in the system be sufficiently large to allow an aggregate table to remain suitable sparse. Ideally each leaf node should provide a table less than 1% dense.

Query Filtering

Before transmitting a query packet to a connection that has provided a query hash table, match the words and URNs in the query against the QHT.

- If any of the lookups based on URNs found a hit, send the query packet
- If at least two thirds of lookups based on words found a hit, send
- Otherwise, drop the packet

It is important to apply the "two thirds" rule only for words. URNs must provide an automatic match.

Consider all text content in the query, including generic search text and metadata search text if it is present. When dealing with simple query language that involves quoted phrases and exclusions, apply the following rules:

- Tokenize quoted phrases into words, ignoring the phrase at this level
- Ignore excluded words – these do not count as table hits or misses
- Remember to apply exclusion to every word in an excluded phrase

See the section on the simple query language for more information.

Gnutella2 Object Search Mechanism

Introduction

The distributed object search mechanism is a very important component of the Gnutella2 architecture. It allows objects distributed throughout the network to be located by a search client in an optimal fashion, requesting and receiving a subset of the total information known about that object.

Model

The Gnutella2 object search mechanism is best described as an "iterative crawl" of the network, with a series of important optimisations derived from the network topology and components:

- A search client iteratively contacts known hubs with its query
- Hubs match the query against the cached hash tables of their neighbours
- Where a table hit occurs, the query is forwarded once only
- The single injected query thus effectively covers the logical hub cluster, which is the basic searchable unit
- Nodes which actually receive the filtered query process it and send results to the search client directly

This model has a number of important advantages:

- Wide network coverage is provided by iterative query injection
- Effective leaf density is increased by querying hub clusters rather than single hubs or even single nodes
- Mutual and two-level filtering and static link compression are leveraged
- The load on each hub cluster is spread equally between its member hubs, providing a larger search base with a lower cost on the aggregation points
- Hubs need a greater inbound bandwidth capacity than outbound bandwidth capacity, which ideally suits asymmetric Internet connections

Search Process Walkthrough

When a search client wishes to execute a new search, the following events happen:

- The search client selects an eligible hub from its global hub cache which has a recent timestamp, has not been contacted more recently than it allows, and has not yet been queried in this search.
- If a query key is not yet available for this hub, a query key request is dispatched.
- Once a query key is available, the search client sends a keyed query to the hub.
- Upon receiving the query, the hub checks the query key for validity.
- The hub then responds with a query acknowledgement packet, containing a list of neighbouring hubs which have now been searched and a list of 2-hop hubs which have not yet been searched.
- The search client adds the list of searched hubs to its "don't try again" list, and adds the list of "try

hubs" to the global hub cache for future selection.

- Meanwhile, the hub examines the query to make sure that it has not received it before. Duplicate queries are dropped.
- The hub then matches the query against the query hash table of all connected nodes, and sends it to those nodes which may be able to field a match.
- While this is occurring, the hub processes the query locally and returns any results it may have.
- Leaves directly attached to the hub which have a potential match will have received the query, and process it locally. They may elect to return results directly to the search client, or may return their results to their hub for dispatch.
- Other hubs in the hub cluster which received the query now examine it to ensure they have not processed it before. They do not send an acknowledgement.
- Assuming it is new, the hubs match the query against the query hash tables of their leaves but not their neighbouring hubs. Potential leaves receive a copy of the query, and the hub processes it locally.
- Once again, the hub returns its own results and may forward results for its leaves if they do not wish to dispatch them directly.
- Meanwhile, the search client receives any search results generated by the hub cluster.
- The search client makes a decision on whether it should continue its search. If so, the process starts again.
- The search client will not requery any of the hubs in the hub cluster, but it has a list of nearby hubs so that the crawl can continue.

Search Security

Introduction

In an environment where a single query injection may result in a number of responses, acting as a virtual "traffic amplifier", it is desirable to verify that the return address is indeed that of the original sender. This prevents the network from being used as a method of generating a large-scale traffic generator.

The Gnutella2 network uses a system of "query keys" to solve this problem.

Before a search client can transmit a query to a particular hub, it must obtain a "query key" for that hub and include it in the transmission. Query keys are unique to the hub generating them and the return address with which they are associated and generally expire after a relatively long period of time.

Search clients request a query key by sending a query key request to a hub, which includes the intended return address for the key. The hub generates a key unique to that return address and dispatches it there. This makes it impossible to get a query key for an IP which is not controlled, and prevents keys from being shared between two nodes (key stealing).

When receiving a query from a foreign node, Gntuella2 hubs check the query key against the one they have issued for the query's requested return address and proceed only if there is a match. If and only if the query key matches will the query acknowledgement be sent, or the search processed locally and forwarded to local leaves and neighbouring hubs.

This has several positive effects. If a query does not have a valid key for the receiving hub, it will not be processed and will thus not generate a traffic amplification effect which may be used in a denial of service attack on a third party host. Secondly, the query key mechanism ensures that queries are only processed if they were transmitted from a host which has control over the host in the return address (in the normal case, this is the same host). This means that flood control mechanisms can remain just as effective as in the TCP case. Similarly, host blocking is possible as the original query source can be verified.

/QKR – Query Key Request

Overview

The query key request asks a destination hub to provide a query key that the requesting node can use to

securely perform queries. This mechanism ensures that the network cannot be used to direct unsolicited packets to external hosts.

Sending

A query key request should be sent to a hub via the connectionless transport if a key is not already available for that hub for the local node address, or if a previously acquired key no longer works or has expired. There is no reason to request a query key via a TCP link.

Receiving

Upon receiving a query key request, a hub should respond with a query key answer (/QKA) packet if it is willing to supply a query key. The answer packet should be dispatched to the return address indicated in the request packet only, and the allocated key should be locked to that address.

/QKR/RNA – Requesting Node Address

Overview

A query key may be requested for a node address other than that of the sender, for example when a UDP-firewalled leaf node wishes to acquire a query key for a remote hub that it will route through a locally connected hub. In this case it must request a query key for the node address of a connected hub.

Sending

This child is required to ensure that the sender knows its address.

Payload

The network address of the requesting node. See datatypes for more information.

Children

This packet has no known children at the current time.

/QKA – Query Key Answer

Overview

The query key answer packet is sent in response to a query key request or in the event that a query with an incorrect or omitted query key is received. It advises the receiver of the correct query key to be used when contacting the sending hub to execute a remote cluster query.

Sending

The query key answer packet should be sent by a hub when a query key has been requested. It should contain a key unique to the local hub and the requesting node address supplied in the request, and should be sent to that address only.

The query key answer packet may also be sent upon receiving a query with a missing or invalid query key. In this case no query should be performed.

Receiving

Upon receiving a query key answer packet, the supplied query key should be stored for later use. If the answer packet contains a sending node address (/QKA/SNA) child which does not match the local node address, it may be forwarded to a connected leaf node via TCP with a matching address. In the forwarding case, a query node address (/QKA/QNA) child is added prior to forwarding to retain the remote (queried) hub's address.

/QKA/QK – Query Key

Overview

The /QKA/QK child contains the query key that was requested.

Sending

This child is required if a key is being issued. Sending a /QKA without a /QKA/QK indicates that the hub does not wish to authorise the requesting node to query it or its cluster.

Payload

A 32-bit query key which is unique to the hub and the sending node address.

Children

This packet has no known children at the current time.

/QKA/SNA – Sending Node Address

Overview

This child specifies the node address for which the key has been issued. This is not necessarily the node that physically transmitted the key request or the query.

Sending

This child is required.

Payload

The node address (with port number optional) of the node which requested the query key.

Children

This packet has no known children at the current time.

/QKA/QNA – Queried Node Address

Overview

If this query key answer packet has been forwarded to a firewalled leaf from a connected hub, a /QKA/QNA child will have been added to indicate the address of the remote hub.

Sending

This child should be added by a hub when relaying a received query key answer on to a leaf via TCP.

Receiving

If this child is present, the query key answer should be treated as if it originated from the stored address rather than the TCP neighbour.

Payload

The node address (with port number optional) of the remote hub.

Children

This packet has no known children at the current time.

Search Descriptor

Introduction

The Gnutella2 object search mechanism defines a powerful query descriptor capable of specifying a number of

different extensible search criteria, and requesting a subset of information about matching objects.

/Q2 – Gnutella2 Query

Overview

The /Q2 packet is the generic Gnutella2 object query descriptor. It contains search criteria, identification, authentication, return addressing and the type of information the search client is interested in.

Sending

A node should send a /Q2 packet to another node when it wishes to execute a query on that node (and its searchable relations).

/Q2 packets should always be filtered against a query hash table if one is available for the outbound link.

Receiving

Upon receiving a query, a node should:

- Verify its authentication
- Send an acknowledgment if it was not received from a hub
- Forward it to connected nodes if necessary (detailed below)
- Process it locally and dispatch results

Forwarding

Forwarding of /Q2 packets is achieved based on fixed rules:

- A query received from a leaf or via UDP is forwarded to all connected nodes
- A query received from a hub is forwarded to connected leaves only

Payload

The payload of a /Q2 packet consists of a GUID which uniquely identifies the search operation.

Children

This packet has many defined child packet types at the current time:

- UDP – Return Address and Authentication
- URN – Universal Resource Name
- DN – Descriptive Name (Generic) Criteria
- MD – Metadata Criteria
- SZR – Object Size Restriction Criteria
- I – Interest

/Q2/UDP – Return Address and Authentication

Overview

The /Q2/UDP child packet specifies the return address to be used for all direct replies, and the query key authentication which authorises transmissions to this address. If the query key is invalid, the query should be discarded and a valid query key issued to the nominated address (if desired).

Sending

This child is required if the query is to be delivered via UDP. It is optional if delivered via TCP: if not included, replies will be reverse forwarded via TCP.

Receiving

If this child is present, all replies should be directed to the nominated address. The query key should be verified before replying.

Payload

A network address (variable length) followed by a query key (fixed length of 32 bits).

Children

This packet has no known children at the current time.

/Q2/URN – Universal Resource Name

Overview

The /Q2/URN child specifies an exact universal resource name which will match the query. Rules:

- If one or more URNs are present in a query, whether or not they are understood, matching must occur on URN only – other criteria should be ignored.
- If multiple URNs are present, only one need match an object for it to be considered a match for the query.
- Following from the above, multiple URNs of the same family may be specified and match multiple objects (this allows a compound query for several known objects to be executed in one packet)

Sending

This child should be added if a specific URN is being sought.

Payload

A string identifying the URN families followed by the URN in either text or binary representation. Text representations should always be 8-bit so that they can be considered binary.

The following compact URN families are recognised:

- **bitprint** or **bp** – 20 bytes of SHA1 followed by 24 bytes of tiger-tree root
- **sha1** – 20 bytes of SHA1
- **tree:tiger/** or **ttr** – 24 bytes of tiger-tree root
- **md5** – 16 bytes of MD5
- **ed2k** – 16 bytes of compound MD4 (ed2k style root hash)

Children

This packet has no known children at the current time.

/Q2/DN – Descriptive Name (Generic) Criteria

Overview

The descriptive name child provides generic query text to be used in matching objects.

Payload

A string in the simple query language.

Children

This packet has no known children at the current time.

/Q2/MD – Metadata Criteria

Overview

The metadata child provides an XML document containing rich query information. The schema identified in the XML must match a metadata schema of prospective matching objects.

Payload

A string containing XML. The <?xml?> header is optional. Standard plural/singular Gnutella metadata schemas are used, for example:

```
<?xml version="1.0"?>
<applications xsi:noNamespaceSchemaLocation="http://www.shareaza.com/schemas/
application.xsd">
<application title="Shareaza"/>
</applications>
```

All string values are in simple query language, including XML attribute and element values.

Children

This packet has no known children at the current time.

/Q2/SZR – Size Restriction Criteria

Overview

The size restriction criteria specify a minimum and maximum size for matching objects.

Payload

Two 32 bit integers, a minimum size followed by a maximum size in bytes.

Children

This packet has no known children at the current time.

/Q2/I – Interest

Overview

The Interest packet advertises an interest in certain information classes, effectively filtering the information that will be returned about matching objects.

Sending

Send an interest packet to restrict the type of information that will be returned about matching objects. For example if querying only to find instances of a known object, advertise interest for "URL" only. Omitting the interest packet produces a default set of response information.

Receiving

If an interest packet is received, only return the requested information types. If no interest packet is present, return all available information that is feasible. Extensions may be withheld.

Payload

An array of strings identifying information classes. The following are defined:

- URL – Locations where matching objects can be found

- DN – Descriptive names of matching objects
- MD – Full metadata of matching objects
- COM – User comments, ratings and reviews of matching objects
- PFS – Partially available objects

Note that default URNs are always supplied and need not be requested.

Children

This packet has no known children at the current time.

Search Acknowledgement

Introduction

When a Gnutella2 hub receives a /Q2 query packet from a search client, it needs to provide the client with acknowledgement for several reasons:

- To tell the search client that the query was accepted, and the hub is alive
- To tell the search client which hubs have now effectively been queried (effectively because the query has been tested against their hash tables, even if this did not result in a forward and local process operation)
- To provide the search client with additional hubs to try if it wishes to continue

Contents

The compact query acknowledgement lists the hubs which have been queried, and provides some hubs which could be queried later.

/QA – Query Acknowledgement

Overview

The query acknowledgement packet is used to inform a search client that a target hub has received its query and is processing it. It also provides information for the search client's hub cache, expanding knowledge of the network and ensuring hubs are not searched more than once in a given query.

Sending

Send a query acknowledgement packet only when operating in hub mode, and only when the query was not received from a hub via a TCP link. Acknowledgements should only be sent after verifying the query key authentication.

Receiving

When a query acknowledgement is received, update the local hub cache as necessary:

- Hubs which have been searched should not be sent this query again
- New suggested hubs should be added and refreshed
- Any "back-off" time should be recorded and honoured.

Payload

A GUID identifying the query being acknowledged.

Children

This packet has several child packet types defined at the current time:

- /QA/TS – Timestamp

- /QA/D – "Done" or Completed Hub
- /QA/S – "Search" or New Hub
- /QA/RA – Retry After
- /QA/FR – From Address

/QA/TS – Timestamp

Overview

The /QA/TS child provides a timestamp representing the current universal time at the sending node. This can be used as a reference when considering other timestamps in the packet, allowing them to be adjusted to eliminate differences between the time setting on the local and remote node.

Sending

This child is optional but recommended.

Payload

A 32-bit integer representing the current UNIX time, or time(NULL).

Children

This packet has no known children at the current time.

/QA/D – Done Hub

Overview

References a hub which has now been queried, and should not be queried again.

Sending

Send a /QA/D child for the local hub and each neighbouring hub, whether or not the hub was actually forwarded the query. If a hub was not forwarded the query, it was still effectively searched as the QHT indicated it could not field a matching object.

Receiving

Upon receiving a /QA/D child, add the hub to the cache, freshen its timestamp and mark it as queried in this search. Do not query this node again for several minutes.

Payload

A network/node address followed by a 16-bit leaf count.

Children

This packet has no known children at the current time.

/QA/S – Search Hub

Overview

References a hub which was not searched in this operation, but could be searched later if it has not already been touched.

Sending

Send a /QA/S child for every unique hub in the local hub cluster which is not a direct neighbour. Optionally select some recent cached hubs and send /QA/S children for them also, to make up at least 10 in total.

Receiving

Upon receiving a /QA/S child, add the hub to the cache and/or freshen its timestamp.

Payload

A network/node address, optionally followed by a 32-bit last-seen timestamp. This makes detecting the address family slightly harder, but the compactness is worthwhile. Timestamps are omitted for hubs in the local hub cluster, which should make up the majority.

Children

This packet has no known children at the current time.

/QA/RA – Retry After

Overview

Advises the search client that the hub does not wish to be contacted again for a specified period of time. Queries earlier than this time may result in a longer-term ban.

Sending

Send this child, with or without executing the search if the hub's resources are becoming saturated. It provides an indirect means of flow control.

Receiving

Upon receiving a /QA/RA child, record the retry after interval and do not send any requests to the hub until the period has expired.

Payload

A 16 bit or 32 bit integer specifying the delay in seconds until the hub may be contacted again.

Children

This packet has no known children at the current time.

/QA/FR – From Address

Overview

This child is used when a query acknowledgement is forwarded from a proxy hub to a firewalled leaf node. It advises the leaf of the original sender.

Sending

When a hub receives a /QA packet for a search originated by one of its leaves, it should forward the /QA to the appropriate leaf after adding a /QA/FR child with the remote hub's network address.

Receiving

If this child is present and the /QA packet was received via TCP from a local hub, the node address within should be used as the remote hub's node address. This information is often reflected in the first /QA/D child, however this is not always present.

Payload

A network address which may omit the port number.

Children

This packet has no known children at the current time.

Search Results

Introduction

Search results from Gnutella2 nodes are delivered in /QH2 query hit packets. Each query hit packet may contain a number of matching object descriptors, and a single query may result in many /QH2 packets from many nodes. Search results may contain varying amounts of information depending on the capabilities of different nodes, and the interest advertised in the original query.

/QH2 – Query Hit

Overview

The /QH2 Gnutella2 query hit packet contains one or more search result descriptors, along with information about the node generating the result set. A single node may generate several /QH2 packets in response to a /Q2 query. Like all Gnutella2 packets, the /QH2 packet is highly extensible.

Sending

Send a /QH2 packet when processing a /Q2 query packet reveals one or more matching objects. Query hits can be returned directly to the /Q2/UDP return address in the query, or routed back along the delivery chain. Firewalled leaf nodes are advised to return /QH2 packets to their connected hub for dispatch, as the hub will be able to make a reliable delivery.

Receiving

Upon receiving a /QH2 query hit packet, check the query GUID against locally initiated queries. If it originated elsewhere, increment the route-back hop counter and forward it to the most appropriate link or UDP endpoint. If it matches a locally generated query, process the search results.

Payload

A single byte representing the hop count, followed by a 16 byte GUID identifying the search that produced these results.

Children

This packet has many child packet types defined at the current time:

- H – Hit Descriptor
- HG – Hit Group Descriptor
- GU – Node GUID
- NA – Node Address
- NH – Neighbouring Hub
- V – Vendor Code
- MD – Unified Metadata Block
- UPRO – User Profile
- BUP – Browse User Profile Tag
- PCH – Peer Chat Tag

/QH2/GU – Node GUID

Overview

Specifies the GUID of the sending node.

Sending

This child is required.

Payload

The GUID of the sending now.

Children

This packet has no known children at the current time.

/QH2/NA – Node Address

Overview

Specifies the node / network address of the sending node.

Sending

This child is optional but recommended.

Payload

The node / network address of the sending node. See datatypes for more information.

Children

This packet has no known children at the current time.

/QH2/NH – Neighbouring Hub

Overview

Lists a hub to which the sending node (a leaf) is connected. This provides a contact point for routing communications to a possibly firewalled leaf node.

Sending

This child is optional, and may appear more than once.

Payload

The node / network address of a hub to which the sending node is connected.

Children

This packet has no known children at the current time.

/QH2/V – Vendor Code

Overview

Identifies the software operating the sending node.

Sending

This child is optional.

Payload

A four character vendor code.

Children

This packet has no known children at the current time.

/QH2/BUP – Browse User Profile Tag

Overview

Indicates that the sending node supports browsing the local user profile.

Sending

This child is optional.

Payload

No payload is defined at the current time.

Children

This packet has no known children at the current time.

/QH2/PCH – Peer Chat Tag

Overview

Indicates that the sending node supports person to person private message chat.

Sending

This child is optional.

Payload

No payload is defined at the current time.

Children

This packet has no known children at the current time.

/QH2/HG – Hit Group Descriptor

Overview

Describes a group of search results.

Sending

This child is optional and may appear more than once.

Payload

A single byte, the group ID number.

Children

This packet has one child packet type currently defined:

- SS – Server State

/QH2/HG/SS – Server State

Overview

Provides a status snapshot of the upload server that applies to search results in the current hit group.

Sending

Add this child to the hit group descriptor to provide server state information.

Receiving

Use this information when interpreting hits linked with this hit group.

Payload

- A 16-bit integer : the current queue length including any active transfers (number waiting plus number transferring)
- An 8-bit integer : the maximum number of concurrent transfers (capacity)
- A 32-bit integer : the speed in kilobits per second for uploads on this server module

Children

This packet has no known children at the current time.

/QH2/H – Hit Descriptor

Overview

Describes a single object which matched the original query.

Sending

Send a /QH2/H child for each matching object to be included in this /QH2 packet.

Payload

No payload is defined at the current time.

Children

This packet has many child packet types defined at the current time, and is often extended:

- URN – Universal Resource Name
- URL – Universal Resource Location
- DN – Descriptive Name
- MD – Metadata
- SZ – Object Size
- G – Group Identifier
- ID – Object Identifier
- CSC – Cached Source Count
- PART – Partial Availability Marker
- COM – User Comment
- PVU – Preview URL

/QH2/H/URN – Universal Resource Name

Overview

Specifies a universal resource name which can be used to identify the object.

Sending

At least one instance of this child is required. Multiple instances are optional.

Payload

A string identifying the URN families followed by the URN in either text or binary representation. Text representations should always be 8-bit so that they can be considered binary.

The following compact URN families are recognised:

- **bitprint** or **bp** – 20 bytes of SHA1 followed by 24 bytes of tiger-tree root
- **sha1** – 20 bytes of SHA1
- **tree:tiger/** or **ttr** – 24 bytes of tiger-tree root
- **md5** – 16 bytes of MD5
- **ed2k** – 16 bytes of compound MD4 (ed2k style root hash)

Children

This packet has no known children at the current time.

/QH2/H/URL – Universal Resource Location

Overview

This child specifies a location where the object being described can be acquired.

Sending

This child should be sent if it was requested and a URL is available.

Receiving

If a /QH2/H/URL child was requested but is not present, the sending node may not have the object. It is legal to advertise information about objects that are not available.

Payload

Empty, or a string which represents an absolute URL.

If a URL is provided, that URL should be considered to point to an instance of the object being described.

If the payload is empty, the sender is indicating that it has the object available locally. It can be retrieved with an HTTP request using the "uri-res" resolver.

Children

This packet has no known children at the current time.

/QH2/H/DN – Descriptive Name (Generic) Criteria

Overview

The descriptive name for the object, and the object's size.

Sending

This child should be sent if it was requested and is available.

Payload

A 32-bit integer indicating the object's size, followed by a string describing it. If a /QH2/H/SZ child is present,

the 32-bit size prefix is omitted here.

Children

This packet has no known children at the current time.

/QH2/H/MD – Metadata

Overview

Provides metadata describing the object.

Sending

This chid should be sent if it was requested and is available.

Payload

A string containing XML. The <?xml?> header is optional. Standard plural/singular Gnutella metadata schemas are used, for example:

```
<?xml version="1.0"?>
<applications xsi:noNamespaceSchemaLocation="http://www.shareaza.com/schemas/
application.xsd">
<application title="Shareaza"/>
</applications>
```

Children

This packet has no known children at the current time.

/QH2/H/SZ – Object Size

Overview

Specifies the size of the object.

Sending

This child should be sent if DN was requested, and if the size is not to be packed into the /QH2/H/DN child.

Payload

A 32-bit or 64-bit integer representing the object size.

Children

This packet has no known children at the current time.

/QH2/H/G – Group Identifier

Overview

Indicates which hit group (/QH2/HG) this hit belongs to. If this child is not present, the hit belongs to group zero.

Sending

Send this child if this hit belongs to a hit group other than group zero.

Receiving

If this child is present, the hit belongs to a hit group other than group zero.

Payload

A single byte, the hit group ID.

Children

This packet has no known children at the current time.

/QH2/H/ID – Object Identifier

Overview

Specifies a 32-bit integer which uniquely identifies the object on the sender's system.

Sending

This child is not normally sent for search results. It is used in browse-user responses to allow objects to be referenced by virtual folder structures.

Receiving

Use this ID to link objects to virtual folder structures.

Payload

A 32-bit integer ID.

Children

This packet has no known children at the current time.

/QH2/H/CSC – Cached Source Count

Overview

Indicates the number of external cached sources known to the sending node

Sending

This child is sent if cached sources are available and interest in URLs was expressed in the query.

Payload

A 16-bit integer.

Children

This packet has no known children at the current time.

/QH2/H/PART – Partial Content Tag

Overview

Indicates that the sending node has only part of the object, and how much it has.

Sending

This child is sent if a /QH2/H/URL child is present and the URL contains only part of the object.

Payload

A 32-bit integer, the number of bytes available.

Children

This packet has no known children at the current time.

/QH2/H/COM – User Comment

Overview

Provides a user review and/or rating for the object.

Sending

This child is sent if a user comment is available, and comments were requested.

Payload

An XML string, of the form:

```
<comment rating="n">
Text
</comment>
```

The rating is an integer 0 <= rating <= 5, indicating the number of "stars". The text may not include HTML, but may include forum-style block codes and emoticon notation.

Children

This packet has no known children at the current time.

/QH2/H/PVU – Preview URL

Overview

Indicates that a preview URL is available for the object. A preview URL will serve a miniature version of the content.

Sending

This child is sent if a preview is available.

Payload

Empty, or an absolute URL.

If a URL is provided, that URL should be considered to point to a preview of the object being described.

If the payload is empty, the sender is indicating that it has a preview of the object available locally. It can be retrieved with an HTTP request to the path:

```
/gnutella/preview/v1?urn:xyz
```

Children

This packet has no known children at the current time.

/QH2/MD – Unified Metadata Block

Overview

Provides metadata describing one or more objects in the packet. This is an alternative method of delivering metadata, which allows metadata for similar objects to be grouped together. It is compatible with the unified metadata block format in Gnutella1.

Sending

This chid should be sent if it was requested and is available.

Payload

A string containing one or more XML documents, each beginning with a <?xml?> header. Standard plural/ singular Gnutella metadata schemas are used, for example:

```
<?xml version="1.0"?>
<applications xsi:noNamespaceSchemaLocation="http://www.shareaza.com/schemas/
application.xsd">
<application index="0" title="Shareaza"/>
</applications>
<?xml version="1.0"?>
<images xsi:noNamespaceSchemaLocation="http://www.shareaza.com/schemas/image.xsd">
<image index="1" description="Shareaza Logo" width="128" height="128"/>
<image index="2" description="Shareaza Screen Shot" width="1024" height="768"/>
</images>
```

An additional attribute, "index" is added to each singular entry. The index is a zero-based integer identifying the Nth /QH2/H child within this /QH2 packet.

Children

This packet has no known children at the current time.

/QH2/UPRO – User Profile

Overview

Delivers a user profile or user profile summary, describing the user operating the sending node.

Sending

Send a version of this child if the operating user wishes to list their screen name in search results.

The current version of this packet contains a summary only, consisting of the screen/nick name. Future versions may include a more comprehensive XML text block in the payload.

Payload

No payload is defined at the current time.

Children

One child packet type is defined at the current time:

- NICK – Nick/screen Name

/QH2/UPRO/NICK – Nick/Screen Name

Overview

Specifies the nick/screen name of the user.

Payload

A string.

Children

This packet has no known children at the current time.

Simple Query Language and Metadata

Introduction

An effective search system must provide a query language which is:

- Powerful
- Intuitive, and
- Natural

At the same time, it is desirable for the language to be reasonably easy to parse in software.

Gnutella2 employs a simple query language that is familiar to users of web search engines and allows most common criteria to be entered intuitively.

Query Language Definition

- Every search string is considered a list of words.
- Words are identified as sequences of alphanumeric characters. Other symbols and white space are ignored.
- In the basic case, every word in the list must appear one or more times in a matching string.
- Words may be marked as negative words or excluded words by prefixing them with a dash (-).
- In this case, every positive word in the list must appear and every negative word must not appear in a matching string.
- Words can be grouped together with quotes. The negation operator (-) may not appear inside a quoted string, but it may prefix a quoted string in which case the negation is applied to the quoted string as a whole.
- The words in a quoted string must appear in the same order in a matching string. Conversely, the words in a negated quoted string must not appear or must not appear in the same order in a matching string.

Examples

Cat Dog

(matches strings with “cat” and “dog”, in any order)

-Cat Dog

(matches strings with “dog” but not “cat”, in any order)

-Cat –Dog

(matches strings with neither “cat” nor “dog”, illegal in an external search as there are no positive words)

“cat dog”

(matches strings with “cat” followed by “dog”, “cat dog” matches, “dog cat” does not)

-“cat dog”

(matches strings without “cat dog”, “cat dog” does not match, “dog cat” does)

“cat dog” –fish

(matches strings with “cat dog” and without “fish”)

Metadata Searching

Searching metadata involves a set of specific rules:

- If a metadata schema is specified as search criteria, matching objects that have metadata must share the same schema
- Metadata can only be compared if criteria and object share the same schema
- Each data member (attribute or element) of metadata is compared separately
- Where a member is specified in the criteria but not in the object, the match fails
- Where a member is specified in the object but on in the criteria, the member is ignored
- The search criteria for each text/string member is in the simple query language defined above
- The search criteria for numeric members is range based, defined in a subsequent section.

Numeric Range Matching

When comparing numeric values, the match function is specified by the search criteria:

- If a value is specified, the value must match exactly
- If a range (X-Y) is specified, the value must lie within that inclusive range
- If (X-) is specified, the value must be greater than or equal to X
- If (-X) is specified, the value must be less than or equal to X

Generic Matching on Metadata

Generic search criteria (/Q2/DN) can be matched against metadata fields, however care must be taken not to match against data members which are not directly descriptive to the object. Client-side schema descriptors are a good solution, listing the scope of a general search on each recognised schema.

HTTP/1.1 Server for Uploading

This is not a core part of the Gnutella2 architecture, and is discussed in depth elsewhere.

HTTP/1.1 Client for Downloading

This is not a core part of the Gnutella2 architecture, and is discussed in depth elsewhere.

User Profile Challenge and Delivery

Introduction

Community is an important part of a peer to peer experience, and a unified user profile format enables meaningful exchange of user metadata. Gnutella user profiles are XML documents adhering to a user profile schema referenced below.

User profile information is exchanged at strategic points in network communications, using packet types defined in subsequent sections.

gProfile Schema

To be copied in.

Profile Challenge and Response/Delivery

Requesting a user profile from another user involves a challenge and response transaction. The challenge step allows the initiator to provide a random token for the respondent to sign with a digital signature and return, verifying its identity. This allows a distributed identity safeguard, which is under development.

/UPROC – User Profile Challenge

Overview

Requests the user profile of a node.

Sending

Send this packet to request the user profile of another node.

Receiving

Upon receiving this packet, respond with a user profile if one is available.

Payload

No payload is defined at the current time.

Children

This packet has no known children at the current time.

/UPROD – User Profile Delivery

Overview

Delivers a user profile.

Sending

Send this packet in response to a user profile challenge (/UPROC)

Payload

An XML string, adhering to the Gnutella user profile schema.

Children

This packet has no known children at the current time.

Michael Stokes

2003-03-26