Dynamic Query Protocol

From Gnutella Developers

Gnutella Dynamic Query Protocol v0.1

Abstract

The traditional Gnutella broadcast search model handles all searches identically -- a search for "mp3" is broadcasted to as many nodes as a search for "Gettysburg Address". As a result, searches for "mp3" return far more results than they need, wasting bandwidth for both query and query hit traffic, while searches for "Gettysburg Address" do not receive the desired number of results. This proposal improves upon this search model by making the query dynamic, with Ultrapeers connecting to more nodes and dynamically adjusting the TTL of outgoing queries along specific connections to use just enough bandwidth to satisfy the query.

Table of contents [hide]

Introduction [edit]

Purpose [edit]

Searches for widely distributed content on Gnutella have traditionally been sent to the same number of nodes as searches for rare content. This resulted in floods of query hit traffic for these searches, forcing Gnutella flow control algorithms to kick in and start dropping traffic. These searches would use a great deal of bandwidth, taking network resources away from searches for rare content that ideally would travel further than they traditionally have. Dynamic querying alleviates this problem by sending searches for widely distributed content to far fewer hosts while sending searches for rare content to as many or more hosts than traditional Gnutella searches. It achieves these goals through changes to the network topology and accompanying changes to the search algorithm.

Requirements [edit]

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in RFC 2119.[1] An implementation is not compliant if it fails to satisfy one or more of the MUST or REQUIRED level requirements for the protocols it implements. An implementation that satisfies all the MUST or REQUIRED level and all the SHOULD level requirements for its protocols is said to be "unconditionally compliant"; one that satisfies all the MUST level requirements but not all the SHOULD level requirements for its protocols is said to be "conditionally compliant".

Architecture [edit]

Overview [edit]

On a high level, this change involves increasing the number of connections that Ultrapeers maintain, decreasing the Time to Live (TTL) of outgoing searches, and sending a search to one Ultrapeer at a time, waiting for results and adjusting the TTL on each iteration based on the number of results received, the number of connections left, and the number of nodes already theoretically reached by the search. When the desired number of results is received, the query ends. Each of these changes is dicussed in detail below.

Ultrapeer Controls all Aspects of Query [edit]

In this scheme, the Ultrapeer controls all aspects of the query. Upon receiving a query from a leaf, the Ultrapeer ignores the TTL of that query and begins a dynamic query. When an Ultrapeer receives a query

from another dynamic-query Ultrapeer, it MUST broadcast that query just as it normally would. When Ultrapeers are performing searches for themselves instead of their leaves, they simply target more results to reflect that they don't have other Ultrapeers searching on their behalf.

High Degree Network [edit]

Clients implementing dynamic querying MUST increase the degree of their Ultrapeer connections, as increasing the degree significantly increases control over the number of nodes hit in a query, maximizing its bandwidth-saving effects. Clients implementing this proposal MUST maintain a minimum of 15 connections to other Ultrapeers, and it is RECOMMENDED that clients increase their degree to 30 or more.

Decrease Time to Live (TTL) [edit]

Clients implementing dynamic querying also MUST decrease the TTL of outgoing queries. One of the biggest problems with the traditional broadcast search model was that a node had little control over the horizon of a query. Once a query was sent, it would attempt to hammer hundreds of thousands of nodes all at once. The combinate of a higher degree network and lower TTLs dramatically reduces this problem because each connection represents a much smaller node horizon. For example, with a degree 6 network where each query is sent with a TTL of 7, a query sent down one connection theoretically would reach 23,436 Ultrapeers (barring cycles in the network graph). In contrast, on a degree 32 network where each query is sent with a TTL of 3, a query sent down one connection theoretically would reach 1,024 Ultrapeers. While the total horizon is also less with a degree 32, TTL 3 network, the point is that increasing the degree and lowering the TTL allows the our algorithm to send the query to far fewer nodes on each iteration, allowing finer- grained control over the query and a more accurate dynamic calculation of what the TTL for the next iteration should be. In choosing the degree and the maximum TTL, clients MUST NOT exceed 200,000 Ultrapeers for their maximum theoretical search horizon -- the number of Ultrapeers they will in hit in searches for the rarest of files. This translates into an overall horizon of 6,000,000+ nodes when taking leaves into account, which should be plenty for any search, and which is only possible on a network performing close to

```
optimally.
```

Life of a Dynamic Query [edit]

This section discusses all aspects of a single dynamic query, including probe queries, search intervals, TTL calculation, etc.

Probe Query [edit]

When a client first begins a dynamic query, it has no way of knowing how common the desired file is. As such, it's unclear what TTL is appropriate for the first connection. To gather more infomation, it is necessary to send a "probe" query that gives some idea as to the popularity of the file. A simple approach is to send a TTL=2 query down 3 connections and wait about 6 seconds for results. If no results, are received, the next query may be sent with a high TTL, such as 3. If the probe returns 30 results, however, it would make sense for the subsequent TTL to be lower, pe perhaps 1. If the client implementing dynamic queries also implements Ultrapeer query routing [2], further optimizations are also possible for the probe. For connections that have supplied query route tables, the dynamic querier MAY choose to check those route tables for hits and to use that information to both determine the popularity of the file more closely and to decide which nodes to send the query to. If, for example, a node has query route tables from 20 connections and 10 of them have hits for the query in their tables, it is clear that the desired content is widely distributed, and it is very likely that the probe alone will satisfy the query. Note that the querier has reached this conclusion before sending a single query down any connection! In this case, the querier may choose to send the query to 5 or more connections that have hits in their tables, knowing that each of them should return results and likely achieving the desired number of results. Here, the query is only sent to nodes that are known ahead of time to contain hits and nodes without hits are not sent the query.

Standard Algorithm [edit]

Once the probe is sent, the standard dynamic query algorithm takes effect. In many cases, the probe will return the desired number of results, terminating the query. When the probe does not achieve the desired number of results, however, the querier must begin the process of dynamically calculating the TTL, per hop wait times, etc. for the remaining connections, as described below.

Calculating the Next TTL [edit]

The algorithm for calculating the TTL for the next connection is one of the most important parts of the dynamic query. This calculation must take into account the number of hosts already queried, the number of remaining connections, and the number of results received. On each iteration, a reasonable algorithm is to divide the number of results still needed by the the average number of results received per Ultrapeer queried to get the number of Ultrapeers you should hit, and then to divide this number by the number of remaining connections to get the total number of hosts to query per connection. With the number of hosts to query per connection, you can easily calculate the appropriate TTL based on the degree of the connection being queried. More formally:

```
Let D = the total number of results desired (150)
Let r = the number of results received
Let R = the number of results needed, or D - r
Let H = the number of Ultrapeers already theoretically queried
Let RH = the number of results per Ultrapeer, or r/H
Let HQ = the number of hosts to query to reach the
         desired number of results, or R/RH
Let C = the number of connections that have not yet
        received the query
Let HQC = the number of hosts to query per connection,
          or HQ/C
Let DEGREE = the degree of the next connection
TTL = the minimum TTL that will reach the desired number
      of hosts for this connection, based on HQC and
      DEGREE
```

HQC and DEGREE can be used to calculate the TTL using the function shown in Section 2.5.2.2. Note that we calculate the TTL using the degree of the connection that will receive the query. For our purposes, we assume that all nodes connected to that node have the same degree -- we don't take into account the likelihood that there is significant heterogeneity in the degrees of the connected nodes because there is no way to know this information and using the degree of the directly connected nodes should give an approximation of the horizon that is good enough for our purposes. The TTL calculation must also take into account the new X-Max-TTL header. The X-Max-TTL header indicates the maximum TTL for fresh queries the connection will accept. As such, the TTL sent along a connection MUST NOT exceed the X-Max-TTL for the given connection. X-Max-TTL is discussed in more detail in the section on headers (Section 2.5.3.2).

Finally, the difference between one TTL and the next is so great in terms of the number of nodes reached that it makes sense to weight this algorithm towards lower TTLs. Given that we are always working with incomplete data, favoring lower TTLs mitigates the danger of choosing a TTL that is too high and wastes network resources.

Calculating the Theoretical Horizon [edit]

At the end of each iteration, the client must add the new number of hosts theoretically queried to the total. For one connection, this calculation is:

```
hosts(degree,ttl) = Sum[(degree-1)^i, 0 <= i <= ttl-1]
```

Note that this only calculates the theoretical horizon for the query, and not the actual horizon. If there are cycles in the network graph, or if the nodes on subsequent hops have different degrees, this number will be innaccurate. Unfortunately, it's the best we can do without imposing overly burdensome restrictions on the overall topology, and this estimate is accurate enough for our purposes. This is the number that should be used in calculating the next TTL.

Time to Wait Per Hop [edit]

After sending a query to one connection, an Ultrapeer MUST wait for hosts to return query hits before sending the next query to the next connection. It must wait for results because those results are vital in determining the popularity of the file, and therefore in calculating the TTL for sending the query down subsequent connections. In experimental observations, 2400 milliseconds per hop is a reasonable latency to expect. So, if a query is sent with TTL=3, the querier would wait 7200 milliseconds for results before sending the query down another connection. While some query hits may arrive after this wait time, 2400 milliseconds per hop is enough to capture the majority of hits that will be returned. As a general rule, the querier MUST wait 2400 milliseconds per hop before sending a query down another connection. Waiting for this period of time is vital because it's otherwise easy for a query to become "out of control", particularly if the content is popular. If the querier does not wait for an appropriate period before sending the query to another connection, the algorithm may incorrectly think that the content is very rare, sending a high TTL query as a result. For queries like "mp3", this is extremely destructive and defeats the purpose of dynamic querying. There are, however, two exceptions to the 2400 millisecond rule. First, for probe queries, it makes sense to send the query down multiple connections simultaneously without waiting for results. Once the probe is sent, however, the querier MUST wait the number of milliseconds appropriate for the hops of the probe. In addition, the querier MAY scale the wait time up depending on how many connections are probed. So, if a probe is sent to 3 connections with a TTL of 2, the querier MUST wait for a minimum of 7200 milliseconds, but also MAY choose to wait for an extra second per connection probed. This is due to the heterogeneity of latencies on the network, and is particularly useful when the querier is taking advantage of query route tables from Ultrapeer query routing [2], where the probe may be sent to 5 or more connections with a TTL of 1. An accurate probe is vital to avoid sending a query with high TTLs for widely distributed content, as should become clear over the course of writing specific dynamic query algorithms. The second exception to the 2400 millisecond rule is an optimization for searches for particularly rare content. When content is rare, waiting for 2400 milliseconds for every hop can make the overall latency of the query unacceptable. If a query is sent to 32 connections at TTL=3, for example, the query would take almost 4 minutes to complete! If a query has taken a significant length of time and has searched a significant number of nodes, the querier can be reasonably sure that the content is at least fairly rare, and MAY choose to become progressively more aggressive with wait times. For example, if the query has theoretically reached 3000 Ultrapeers and has returned under under 10 results, the querier MAY choose to decrease the per-hop wait time by 100 milliseconds for the next iteration, or even by an ever-increasing number of milliseconds depending on the iteration index. Decreasing the wait time by 100 milliseconds on ten successive iterations will decrease overall latency significantly. While this optimization is valuable and even recommended, implementors must take care not to be overly aggressive. This is particularly important for content in the "middle range" -- content that is not too common, but not too rare. The danger with this type of search is again that the algorithm will select overly aggressive TTLs and waste network resources, and implementors should determine appropriate values through experimentation.

Query Termination [edit]

There are several conditions that will terminate the query. These are:

1. The client receives 150 results for an Ultrapeer-initiated query, or 50 results for a leaf-initiated query.
2. There are no connections left to query.
3. The theoretical horizon has hit the 300,000 limit.

The number of results to search for is covered in further detail in the "Number of Results" (Section 2.5.3.1) section. Algorithm authors also MAY choose to impose other restrictions on the query to safeguard against attack or abnormal conditions. For example, a client MAY wish to limit the total lifetime of the query to, say 3 minutes.

Other Considerations [edit]

Number of Results [edit]

Clients MUST target a maximum of 150 total results for a given query. This many results will give the client enough sources to successfully download the file while saving network resources on searches for popular content. When 150 results are received, clients MUST NOT send the query to any more hosts. Because most queries are initiated by leaves, and because most clients connect their leaves to 3 Ultrapeers, each Ultrapeer MUST attempt to get only 50 results for a query received from one of its leaves.

New Connection Headers [edit]

There are several new connection headers that clients MUST add to indicate their support for these features. Each header is outlined below.

Maximum TTL [edit]

Clients MUST include a new "X-Max-TTL" header in their handshakes. This header indicates that we should not send fresh queries to this connection with TTLs higher than the X-Max-TTL. If we are routing traffic from other Ultrapeers, the X-Max-TTL is irrelevant. The X-Max-TTL MUST NOT exceed 4, as any TTL above 4 indicates a client is allowing too much query traffic on the network. This header is particularly useful for compatibility with future clients that may choose to have higher degrees but that would prefer lower TTL traffic from their neighbors. For example, if future clients connect to 200 Ultrapeers, they could use the X-Max-TTL header to indicate to today's clients that they will not accept TTLs above 2. A typical initial value for X-Max-TTL is 3, as it the following example:

```
X-Max-TTL: 3
```

Ultrapeer Degree [edit]

The X-Degree header simply indicates the number of Ultrapeer connections this nodes attempts to maintain. Clients supporting this proposal must have X-Degrees of at least 15, and higher values are preferable. This header takes the following form:

```
X-Degree: 32
```

Dynamic Query Version [edit]

Finally, the client MUST indicate the version of dynamic queries supported. This will allow future versions to make connections only to hosts supporting newer dynamic query features. This header takes the following form:

```
X-Dynamic-Querying: 0.1
```

Conclusion [edit]

This proposal addresses one of the most glaring flaws in the traditional Gnutella search model, namely that searches for widely distributed content were handled in the same way as searches for rare content. This change uses much less bandwidth for searches for common files while doing more work for searches for rare files, improving overall network performance considerably.

References [edit]

[1] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, March 1997.

[2] Fisk, A., "Gnutella Ultrapeer Query Routing, v. 0.1", May 2003, [1] (*http://f1.grp.yahoofs.com /v1/EHbCPh8iRqHKdls98hCO8-XiZzjgeUSUnbMC7ifpIejfpUIlFs5A7E6V4-miCr5-PdWwmDsEGHkyGcWR /Proposals/search/ultrapeer_qrp.html*).

[3] Rohrs, C. and A. Singla, "Ultrapeers: Another Step Towards Gnutella Scalability", December 2001, Ultrapeers_proper_format.html (*http://groups.yahoo.com/group/the_gdf/files/Proposals/Ultrapeer/*).

[4] Osokine, S., "Search Optimization in the Distributed Networks", December 2002, [2] (*http://www.grouter.net/gnutella/search.htm*).

Author's Address [edit]

Adam A. Fisk
LimeWire LLC
EMail: afisk@limewire.com
URI: http://www.limewire.org

Appendix A. Acknowledgements [edit]

The author would like to thank the rest of the LimeWire team and all members of the Gnutella Developer Forum (GDF).

Retrieved from "http://www.the-gdf.org/wiki/index.php?title=Dynamic_Query_Protocol"

Views

- Article
- Discussion
- Edit
- History

Personal tools

- Create an account or log in

Navigation

- Main Page
- Community portal
- Current events
- Recent changes
- Random page
- Help

Search

Go Search

Toolbox

- What links here
- Related changes
- Special pages

MediaWiki
Attribution

- This page was last modified 16:17, 1 Apr 2005.
- This page has been accessed 205 times.
- Content is available under Attribution.
- About Gnutella Developers
- Disclaimers