

# Gnutella-Pro: What bandwidth barrier?

Richard Massey, Shriram Bharath & Ankur Jain  
{*rmassey,sbharath,ajain*}@cs.ucsd.edu

## 1. Introduction

This paper discusses the history, and evolution of the peer-to-peer file sharing network protocol Gnutella. It highlights the disadvantages of Gnutella particularly with respect to network bandwidth wastage due to excessive *pings*, *pongs* and *pushes*, and suggests changes to the protocol that have yielded very promising results in reducing the number of pings and pongs on the network. Our simulations show that with a few modifications to the protocol, the overall efficiency in the usage of the network increases significantly. In section 2, we detail the original Gnutella protocol and point out its inefficiencies. In section 3, we mention our suggested improvements to the protocol, and finally in section 4 we detail and analyze our simulation results.

## 2. Gnutella Protocol Specification

Gnutella is a protocol designed for distributed search. It differs from other peer-to-peer protocols like Napster in that unlike Napster, it is essentially a decentralized model, though it is capable of supporting the traditional client/centralized server model as well. In Gnutella, every client is a server, and vice versa. Users issue queries through these clients and at the same time, also accept queries from other similar clients searching for matches for files against those existing in their own file systems. The great advantage of Gnutella spawns from its distributed nature – it is highly fault-tolerant – its operation will not be disrupted if a client (also referred to as a *servant*) or a set of clients, go offline.

### 2.1 Protocol Definition

The Gnutella protocol defines how clients communicate with each other over the network. The protocol consists of five descriptors that are used for this communication and a set of rules governing how these clients exchange these descriptors. The descriptors that Gnutella uses are:

*Ping*: This is used to discover hosts on the network as each node receiving a ping is expected to respond with one or more pongs.

*Pong*: This is the client's response to a received ping, and includes the client's IP address and its file sharing details.

*Query*: This is the mechanism used to search for files in the Gnutella network. If a client has the required file, then on receiving the Query, it responds with a QueryHit.

*QueryHit*: This is the response sent by a client that finds a match to the query in its own data sets. It includes information that the initiator of the query can use to acquire the matching request.

*Push*: This descriptor allows a firewalled client to participate in the file sharing.

## 2.2 Gnutella client communication

A Gnutella client connects itself to the network by establishing a TCP/IP connection with another client currently already on the network. The address of this client is assumed to be well known and its acquisition is not part of the protocol specifications. Once the connection has been built, the client begins communicating with other Gnutella clients (servers) through the previously defined descriptors. These descriptors are preceded by Descriptor Headers that have the following fields – a Descriptor ID, a unique 16-byte string identifying the descriptor on the network, a Payload Descriptor, which says which of the five possible descriptors it is, a TTL, which indicates how many times the descriptor packet will be forwarded before it expires, Hops, which indicates the number of times the descriptor has been forwarded, and the Payload length, the number of bytes of data following the header.

The newly joining Gnutella client actively probes the network for other clients by sending pings to all of its connections. The client is not restricted in the number of pings it sends, and the responsibility of minimizing the traffic in the network falls solely on the client implementer. When a server receives this ping, it returns a pong with all the relevant details included, to the node that initiated the ping. It then updates the TTL and Hop count fields of the packet and if the TTL is greater than zero, it forwards the packet to all the connections that it has maintained.

When initiating a query, the Gnutella client includes in the descriptor packet the search criteria and the minimum speed that the responding client must have. The search string is a *null* terminated string that has a maximum length bounded by the payload length field of the descriptor header. If a server on the network satisfies both the minimum speed requirement and contains in its dataset a match to the query, it sends a QueryHit descriptor that includes besides the speed and the hit details, the port number on which it will respond and its IP address. Likewise, every server that satisfies the requirements specified by the client and contains a matching file, returns a QueryHit. All these hits are composed into a result set that the client can go through and decide which server to commence the download from. A subtlety to be noted here is that, if the client generates multiple queries, it can identify which response matches which query by looking at the Descriptor ID field in the responses.

If a Gnutella server/client is behind a firewall, then too it will generate the same kind of responses that it would have generated had it not been so. The difference is that the client that initiated the search will never be able to build a TCP connection to the server and so it will never be able to download the file. However, the transfer would be possible from the server, if the server itself builds the same TCP connection to the client. But how does the server know that it is expected to build this TCP connection? To get around this problem, the client uses the fifth Gnutella descriptor, the push. If the Gnutella client is unable to access the file that it wants from the server, then it generates a push request that it forwards across all its links exactly like the ping. Each node that receives this push checks if it is meant for itself and if not, it simply forwards the packet along all of its links. If it finds that the packet is meant for itself, it builds a

TCP connection to the client that initiated the push and proceeds to transfer the requested file. These are all validated using the Descriptor ID.

### **2.3 Shortcomings of Gnutella**

The Gnutella protocol is very simple in its specifications and as highlighted above, owing to its distributed nature is highly stable. However on closer analysis, we see that the network usage is anything but efficient. The current Gnutella protocol pings every server in the network, thus effectively flooding the network, as the growth rate of the number of pings in the network is exponential. The current protocol has no restriction on the TTL. If the average connectivity of the network is assumed to be 'n', then the number of pings in the network varies as  $n^{(TTL)}$ . If every node in the network has a T1 or a T3 connection to the network, this wouldn't be such a bad idea, but the fact of the matter is that this is not true. So, nodes with lower bandwidths, like the 56K users effectively slow down the network considerably by taking up more connections than they can handle optimally.

Another shortcoming of Gnutella is the push descriptor. No node actually knows that it is behind a firewall. Also it has no idea if a server that it is trying to connect to, via a TCP connection, is behind a firewall. All the client does is wait for a period of time for the server it is trying to connect to respond. If the server doesn't reply, the client assumes that the server is behind a firewall, and so initiates a push. However, there may have been other reasons why the server hadn't responded – it could have been busy with all of its other connections, or perhaps the client itself is behind a firewall. But all the client does is push with an associated timeout. If the descriptor times out, the client initiates another push. This continues till the server responds or the client manually terminates the connection request. Obviously the number of push requests is going to be even larger than the number of pings leading to a very wasteful usage of the network.

Besides this bandwidth wastage, there is also a problem of authentication. The client can always be a malicious one that pushes a return address of some other client. The server, knows only where to send the requested file and then proceeds to build the TCP connection to the wrong client and start sending the file. The malicious client could initiate sending a huge file that would effectively congest the victim's network, and bring it down.

### **3. Gnutella-Pro**

We suggest changes that address the above-mentioned shortcomings of Gnutella. First is the problem of network wastage and eventual network congestion owing to the excessive number of pings, particularly those routed through nodes that have a network connectivity far exceeding the optimum for that bandwidth. What we suggest is that the maximum connectivity of a node in the network be restricted by the nodes bandwidth connection to the network. For example, a 56K user could be restricted to just 2 TCP connections whereas a T3 user could be allowed up to 15. The numbers are not representative of anything but merely present a guide. This will make sure that a node never has more connections that it can manage at an optimal speed and thus in a way prevent network congestion. Which nodes are selected is decided randomly.

If a node receives a ping, it first checks to see if it is allowed to open up more connections. If it is, then it sends a pong back to the initiator of the ping and then forwards the ping along two of its links. If however, it finds that it cannot accept any more connections, then it does not return the pong, and merely forwards the pings. The rest of the functionality remains the

same and irrespective of whether the node that receives the ping returns a pong or not, it updates the TTL and Hop count fields in the descriptor.

The other problem with the Gnutella protocol as discussed above, is the ‘push’. Since the client receives the pong from the server that is behind the firewall, it also has the node through which the pong was routed to it. In this way, each node knows from where it received that packet. So, using this *next-hop* method, the client can actually push the request along the exact path that the pong was routed all the way back to the server. This saves us the cost of flooding the network with pushes. This however, won’t work for the case where both the client and the server are behind firewalls. In that case, there is no way in which the connection can be built.

The problem of authentication of the user and thus prevention of the denial-of-service attacks can be tackled by using a *handshake*. Prior to the server sending data to the client, it builds the TCP connection and confirms that the client had indeed requested the file that is about to be sent. The client will respond with the GUID of the query if it actually wishes the data to be sent. DoS attacks become impossible because the target of the attack will not respond with the GUID the server is expecting, and the server will not send the file.

### 3.1 Protocol Specification

Gnutella-pro is much like the old Gnutella, but was designed not to waste any bandwidth. Many of the messages contain the same data, but have been compacted to use the least number of bytes possible. As few messages as possible are sent, but we do not remove any messages such that Gnutella-pro loses functionality as compared to Gnutella. The only small exception is that in Gnutella-pro, a client is unable to tell how many files are being shared on the network.

#### Routing

*Ping:* Ping message are forwarded to 2 randomly chosen neighbors in the overlay. A Pong response is only sent if the node is ready to open a new connection.

*Pong:* Pong messages are routed along the Ping path.

*Query:* Queries are forwarded to all neighbors.

*QueryHit:* QueryHit messages are forwarded on the corresponding QueryHit path.

*Push:* Push message are forwarded on the corresponding QueryHit path.

#### Message Definitions

*Header:* Gnutella: 0 bytes / Gnutella-pro: 0 bytes

GUID	Function	TTL	Hops	Payload Length
6-Bytes	1-Byte	1-Byte	1-Byte	2-Bytes

*Ping:* Gnutella: 0 bytes / Gnutella-pro: 0 bytes

Empty
-------

0-Bytes

*Pong:* Gnutella: 14 bytes / Gnutella-pro: 6 bytes

IP Address	Port
4-Bytes	2-Bytes

*Query:* Gnutella: Avg. 13 bytes / Gnutella-pro: Avg. 12 bytes

Minimum Speed	Search Text
1-Byte	Variable (11 byte average*)

*QueryHit:* Gnutella: Avg. 243 bytes / Gnutella-pro: Avg. 182 bytes

# of Hits	IP Address	Port	Speed	Results
1-Byte	4-Bytes	2-Bytes	1-Byte	Variable (182 byte average**)

*Push:* Gnutella: 26 bytes / Gnutella-pro: 16 bytes

GUID	IP Address	Port	File Index
6-Bytes	4-Bytes	2-Bytes	4-Bytes

\* All averages given for variable length fields were determined by analyzing sample data from the existing Gnutella network.

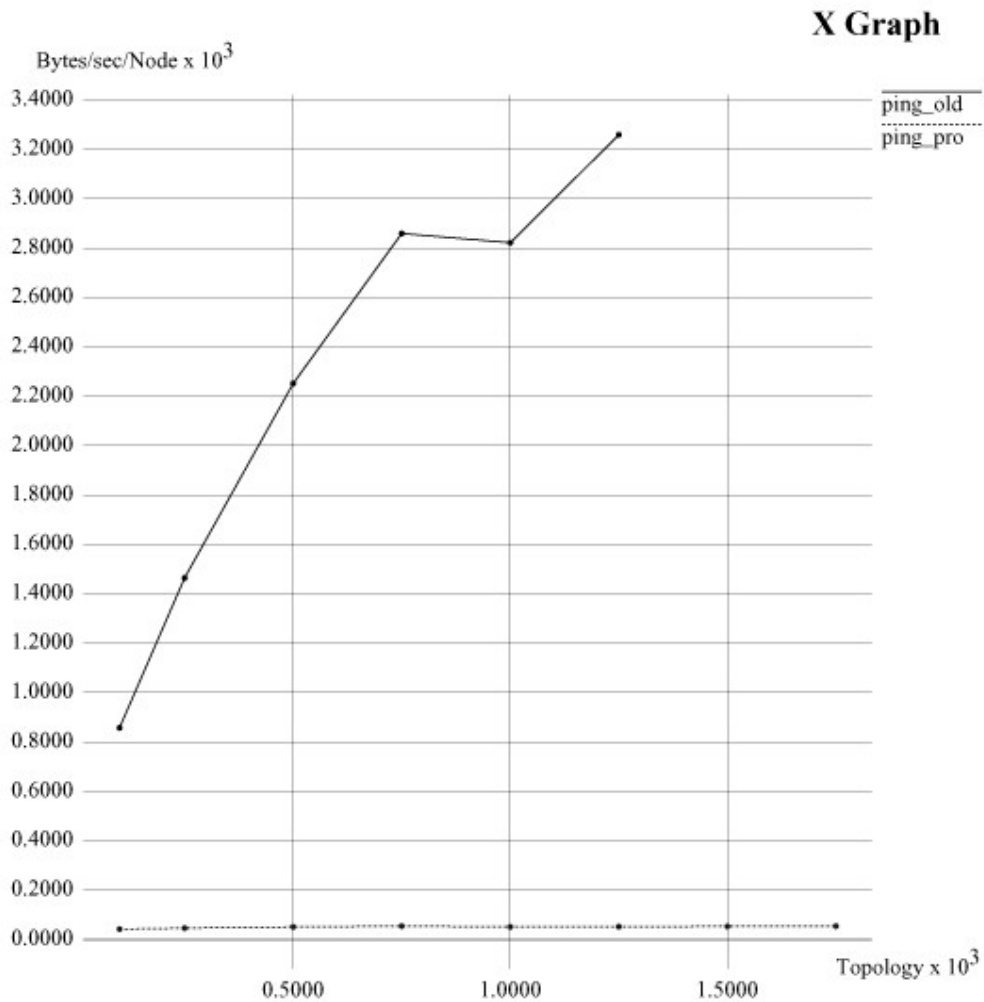
\*\* This average represents a compressed version of the average results field we viewed on the Gnutella network. The results field can contain only a limited set of ASCII characters, which in the worst case can be converted to use 6-bits per character. We are exploring possible encoding schemes that could reduce this further.

#### 4. Simulations and Analysis

We simulated the Gnutella and the Gnutella-pro protocols in *ns*. The results of the comparison are staggeringly in favor of Gnutella-pro but this did not really surprise us too much. The original Gnutella protocol was always inefficient and so fitting it with our suggested modifications was definitely going to up its performance by quite a bit.

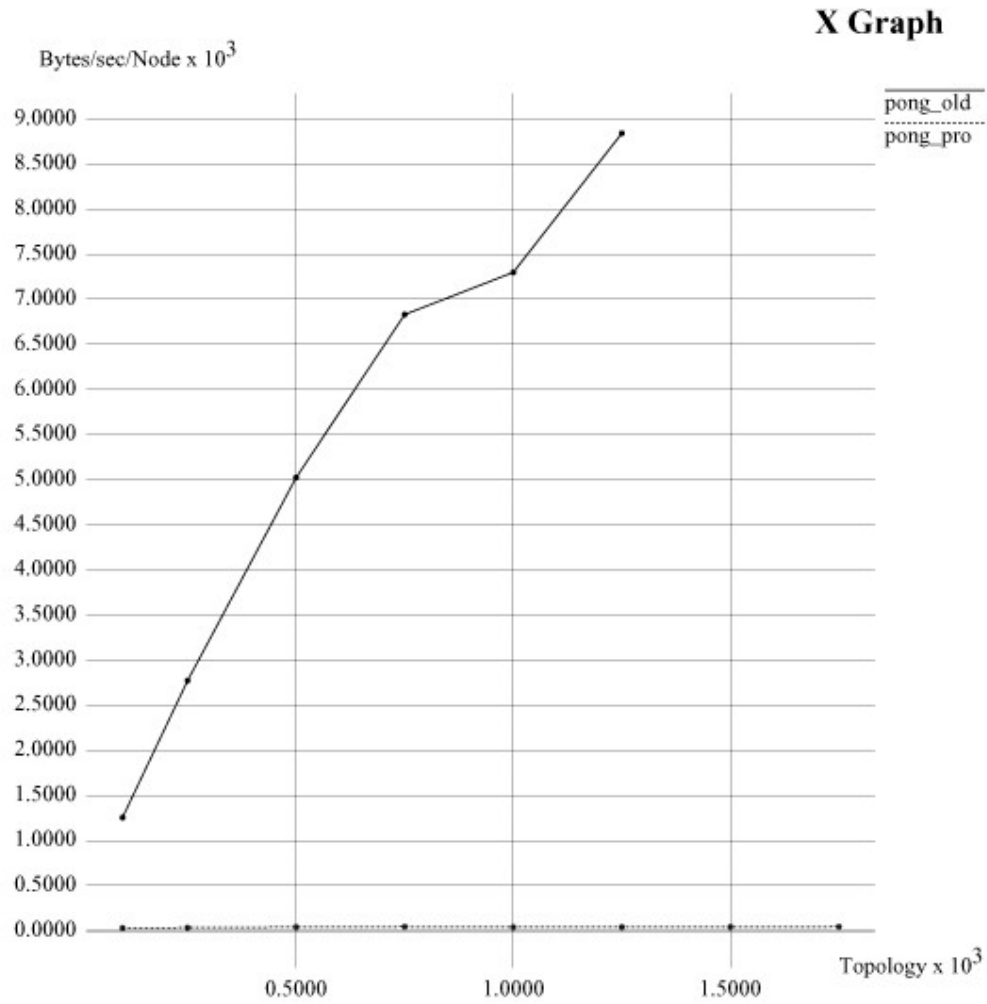
We modeled each node of the Gnutella overlay as a node in *ns*. Links are randomly added between nodes to simulate Gnutella users connecting. A set of random timers are used to simulate user actions such as querying and downloading. Constants used in our timers were chosen by analyzing network traffic from the existing Gnutella network. Our analysis showed that the average user queries once per minute, and had an average search string size of eleven. Our simulation of the original Gnutella had to be limited to topology sizes of 1250 or less. The reduction of messages in Gnutella-pro allowed us to simulate up to 1750. However, the performance trends are easily identifiable even with small topologies.

Using data from [2], we build up our networks so that they would have similar characteristics to the real Gnutella network. 30% of the hosts share most of the files, but the other 70% do return hits on occasion. Each topology is generated randomly, but on average contains about 60% dialup users, 30% Cable/DSL users, 5% T1 users, and 5% T3 users. Each node is given a maximum degree based on how much bandwidth it has available, and nodes with higher bandwidth leave the network less frequently. Dialup users are connected on the order of hours, Cable/DSL users on the order of days, and T1/T3 users for weeks.



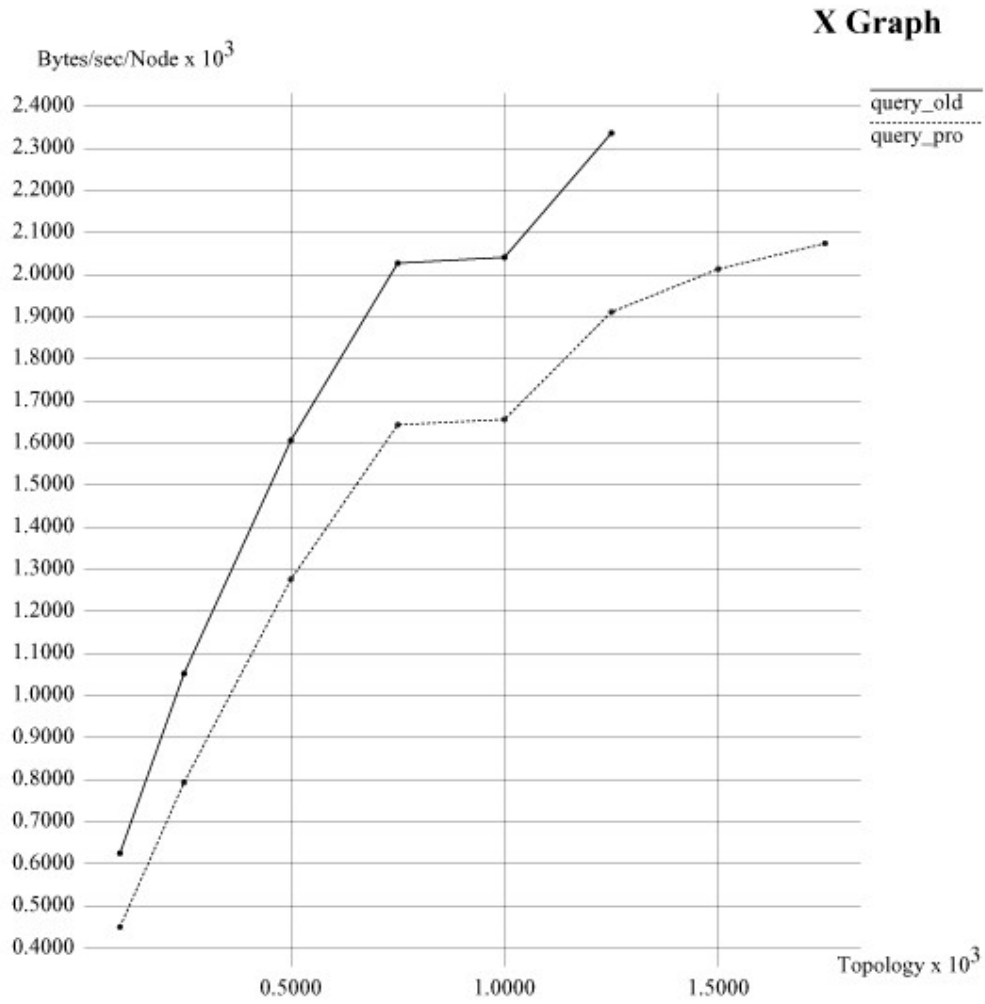
**Figure 1 – Ping Traffic Comparison**

Figure 1 shows the comparison of the number of bytes of ping packets doing the rounds on the network per node per second. As is evident from our earlier description, Gnutella floods the network with pings. This flooding causes the amount of ping traffic to increase exponentially. Gnutella-pro controls the number of pings over the network and so the performance is vastly superior. In Gnutella-pro, the amount of ping traffic increases linearly with the size of the network, and so it is roughly constant per node.



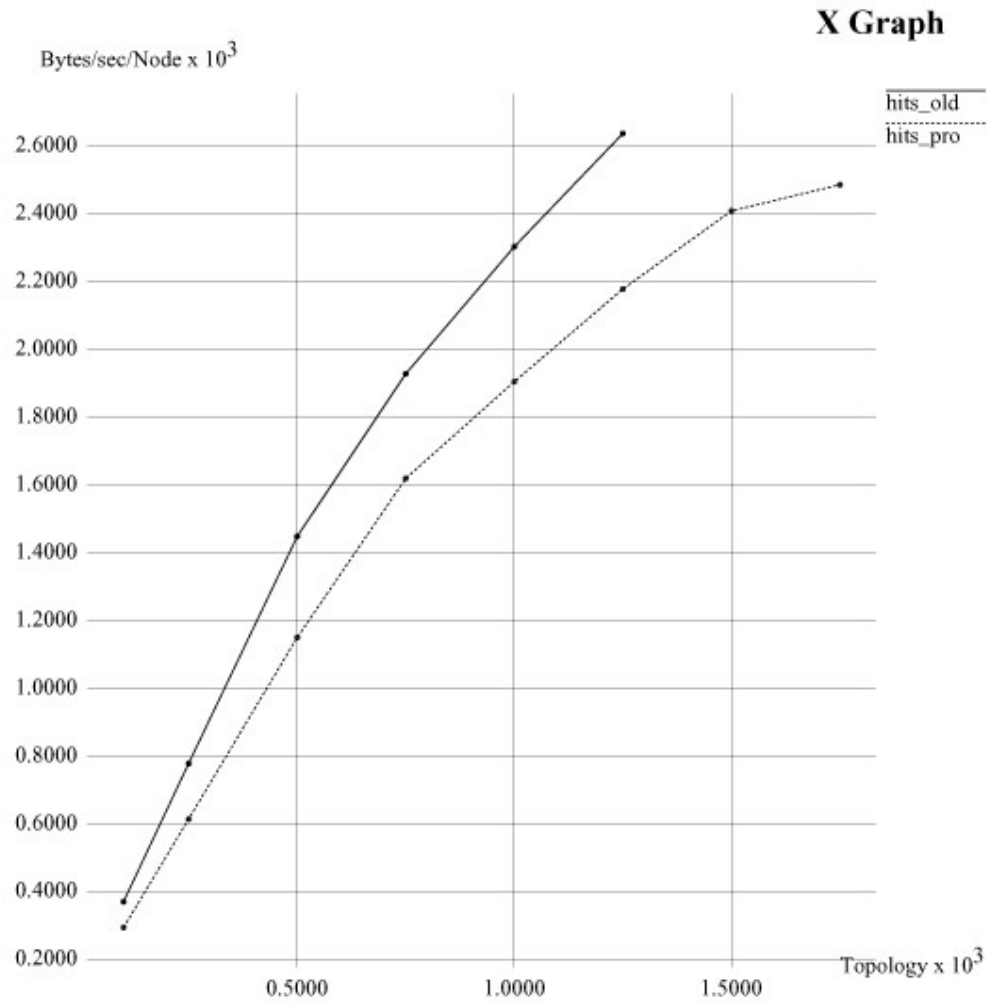
**Figure 2 – Pong Traffic Comparison**

Figure 2 shows the same comparison for pong traffic. As expected, the pong traffic matches that of the ping traffic. We see Gnutella increasing exponentially, and Gnutella-pro remaining constant.



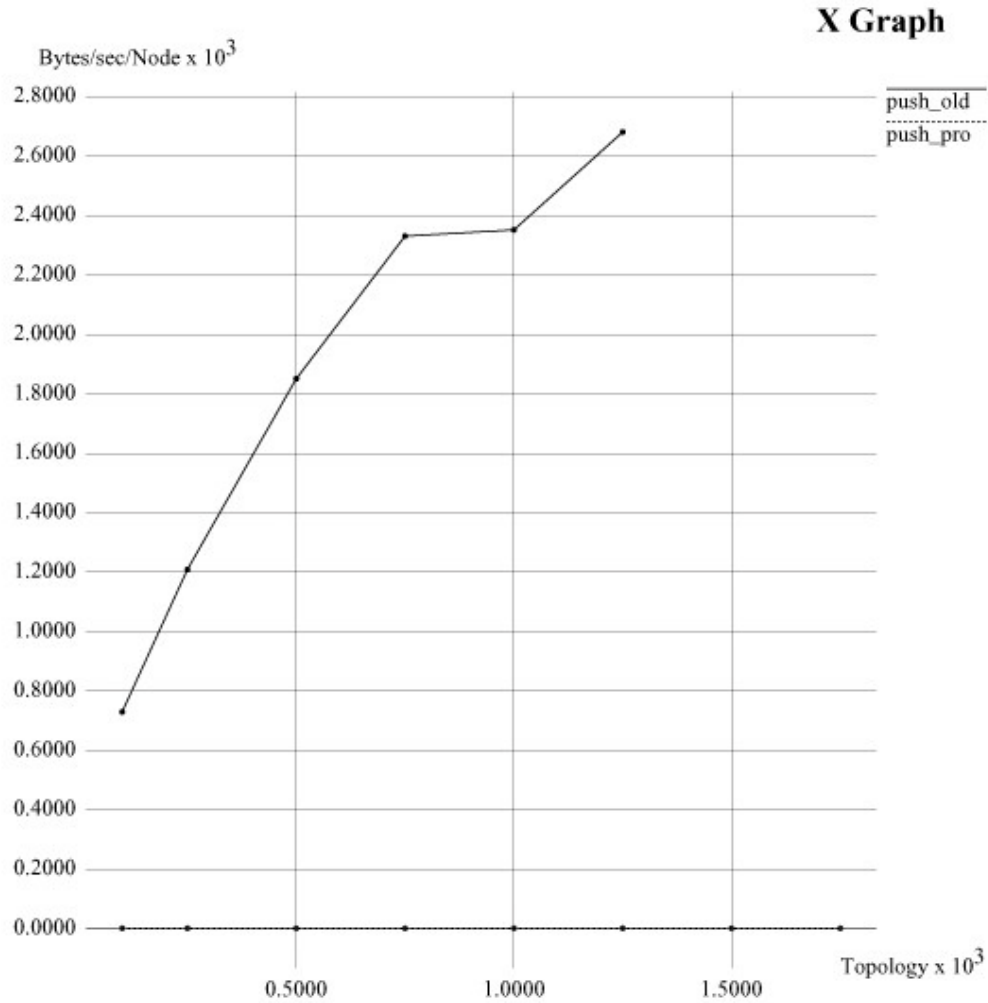
**Figure 3 – Query Traffic Comparison**

In Figure 3, our increase in performance doesn't come from the forwarding mechanism. It is a result of Gnutella-pro's more efficient use of space in packet headers. The difference is fairly significant, showing just how inefficient the original header was.



**Figure 4 – Hit Traffic Comparison**

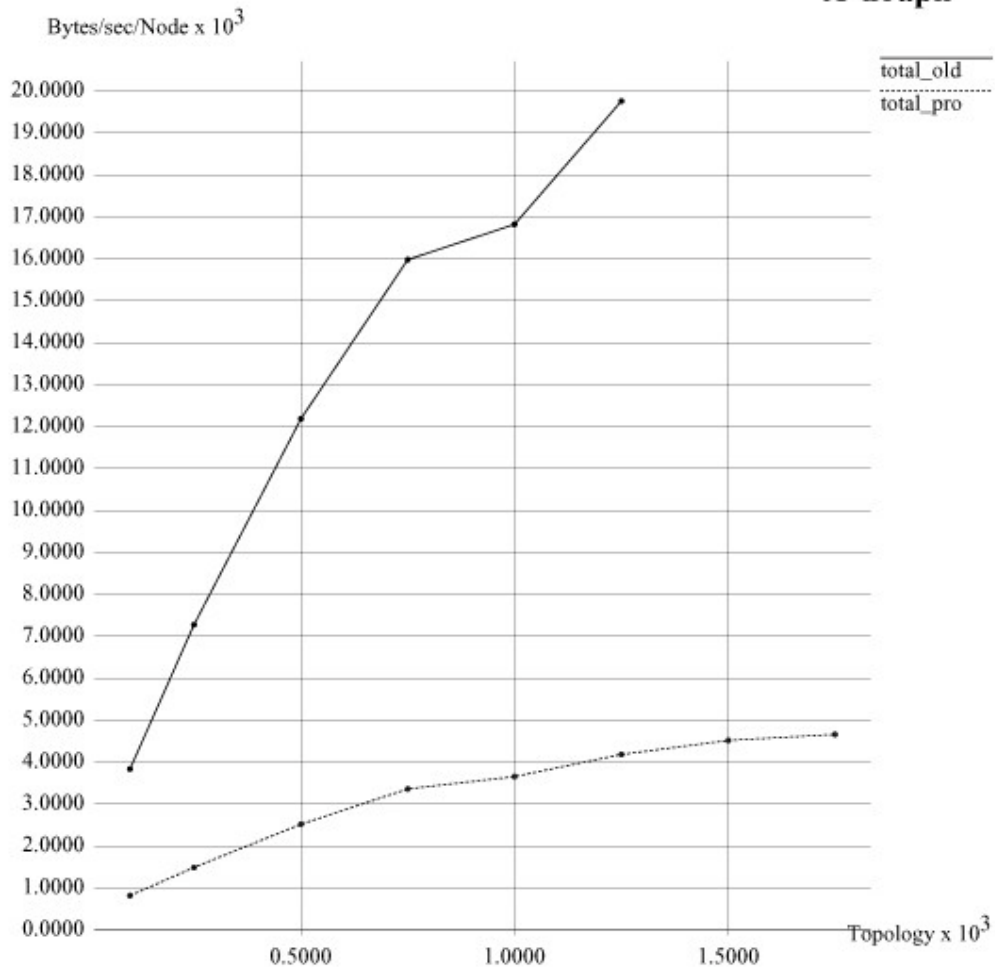
In Figure 4, we also see a reduction in traffic caused by more efficient use of headers. It is also reflecting the compression of text strings in the result portion of the packet.



**Figure 5 – Push Traffic Comparison**

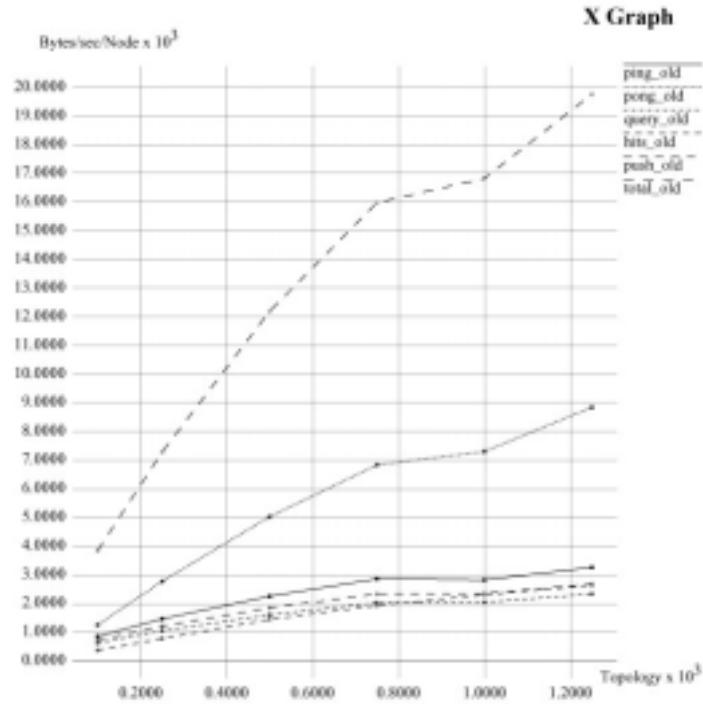
Figure 5 shows how efficient it is to route push requests as compared to flooding them. This is a comparison against the original Gnutella specification. Many existing clients have already implemented this functionality for obvious reasons.

## X Graph

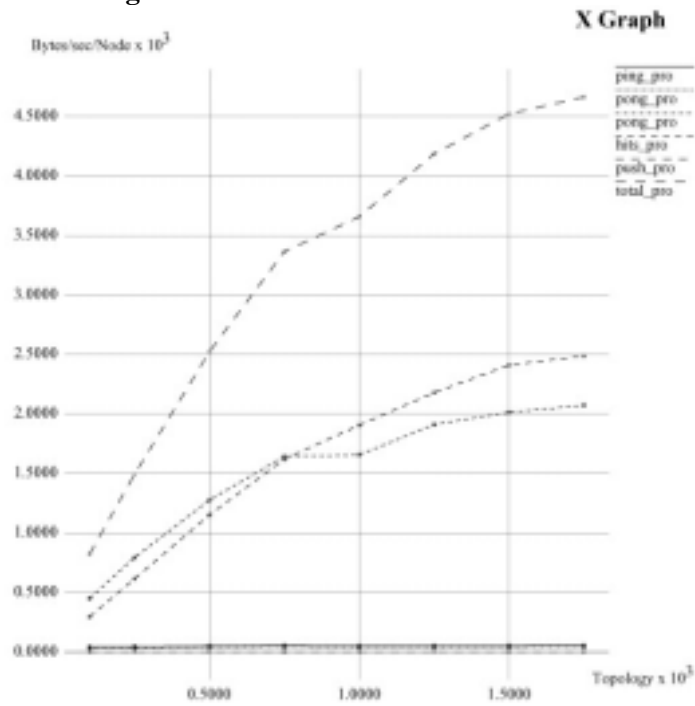


**Figure 6 – Total Traffic Comparison**

Figure 6 shows the sum of the previous 5 graphs. It is the best comparison of the overall performance of each protocol. Gnutella-pro is obviously using less bandwidth, and the trend appears to be that it is growing more slowly as the size of the network increases.

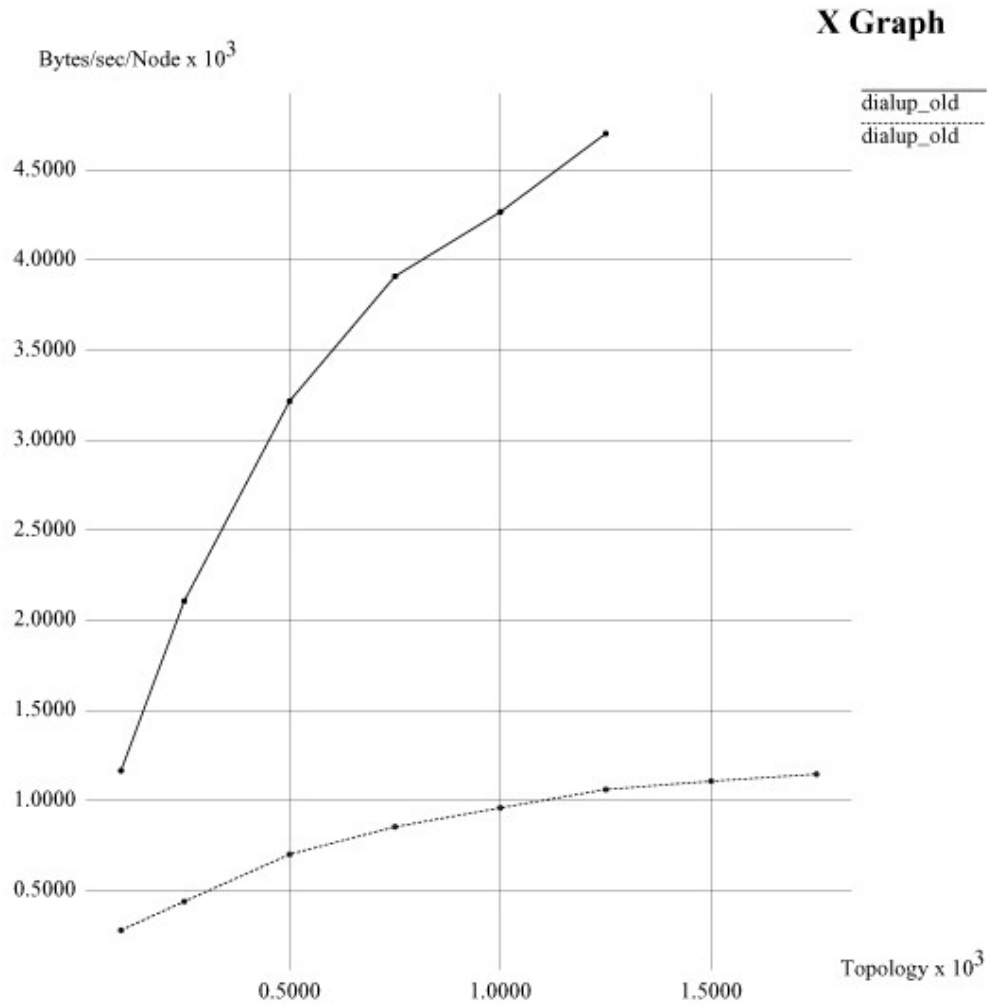


**Figure 7 – Gnutella Traffic Breakdown**



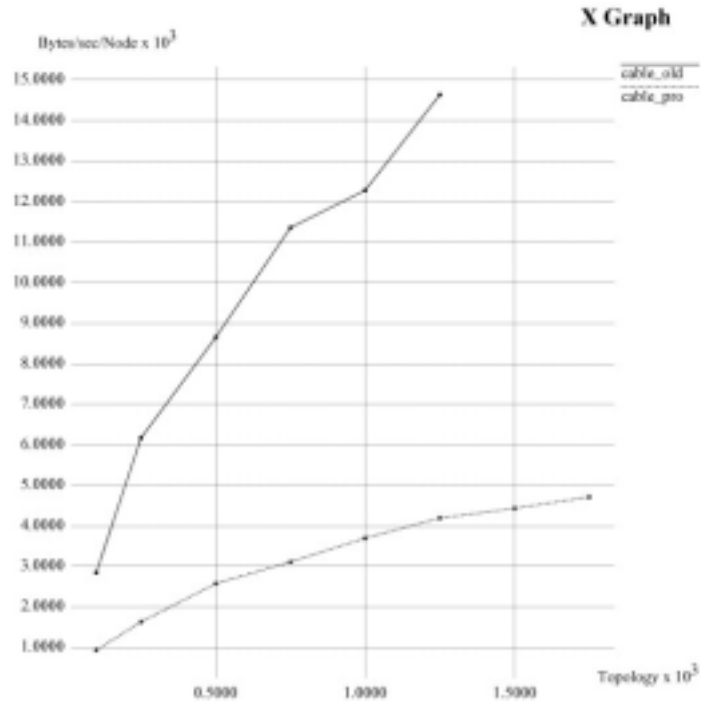
**Figure 8 – Gnutella-pro Traffic Breakdown**

Figures 7 and 8 show a breakdown of bandwidth consumption of each message type.

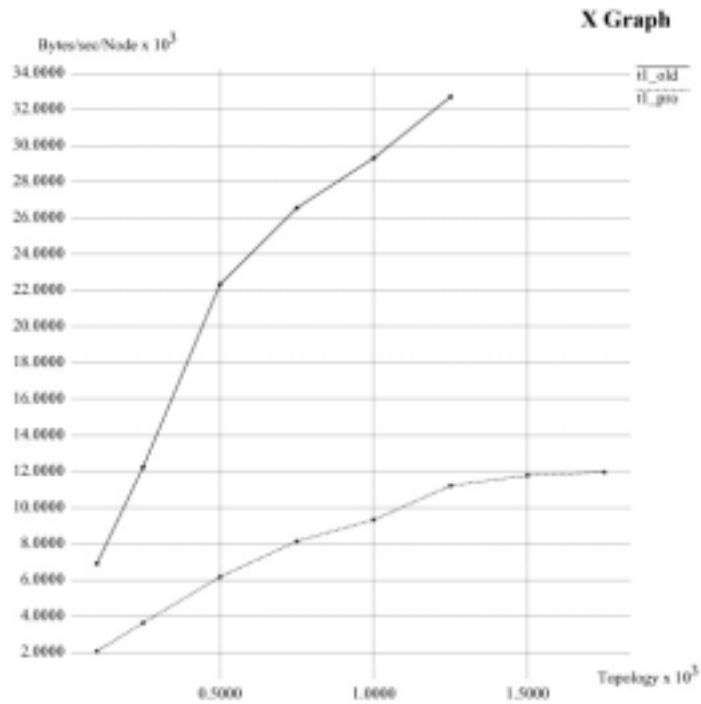


**Figure 9 – Dialup Traffic Comparison**

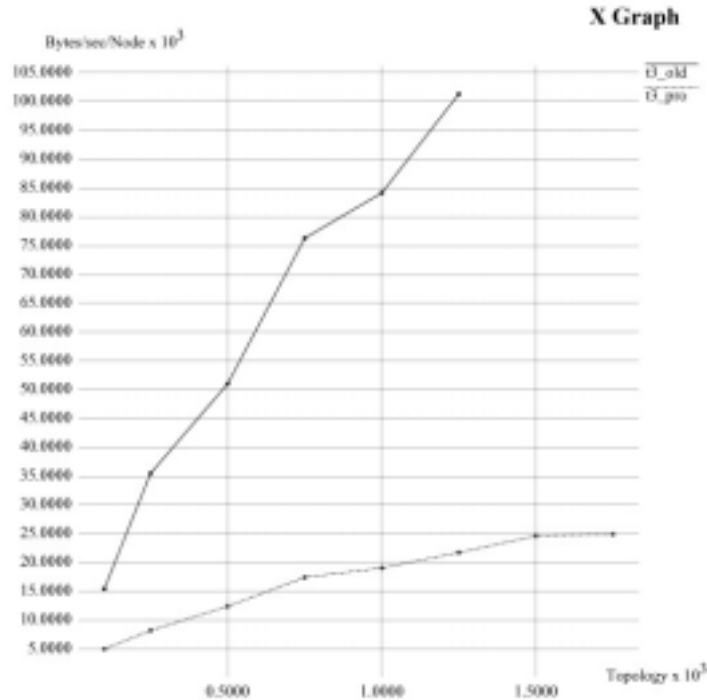
Figure 9 shows the average bandwidth used by nodes with 56KB dialup connections. The bandwidth required of these users is the so-called “Bandwidth Barrier”. Here, Gnutella-pro really proves that it is more scalable than the original Gnutella protocol. At our maximum simulated topology size of 1750 nodes, Gnutella-pro consumes 1.2KB of a dialup users bandwidth. Unfortunately, we were unable to simulate larger topologies so we can only speculate at when Gnutella-pro would reach the “Bandwidth Barrier”. The trend however suggests that Gnutella-pro can increase to fairly large network sizes.



**Figure 10 – Cable Traffic Comparison**



**Figure 11 – T1 Traffic Comparison**



**Figure 12 – T3 Traffic Comparison**

Figures 10, 11, and 12 show the same analysis as Figure 9, but for nodes with Cable, T1, and T3 connections. As expected, Gnutella-pro uses less bandwidth in all cases. These results were fairly surprising to us, as even higher speed connections give up a large percentage of their bandwidth to Gnutella traffic. In the original Gnutella, a T3 user can be transiting as much as 100KB/sec in a network of size 1250. Our conclusion from these graphs, is that even users with faster links have to carefully consider the number of connections they choose to maintain.

## 5. Conclusion

In our simulations of Gnutella-pro, we were able to show that it is possible to achieve the same functionality of Gnutella with significantly less bandwidth usage. The fact that we could simulate Gnutella-pro with up to 1750 nodes, as compared to only 1250 for Gnutella, shows that Message complexity of Gnutella-pro is significantly less. So finally we conclude that using an efficient protocol, and properly assigning node degrees, we can extend the life of Gnutella far beyond its current limitations.

## References

- [1] Gnutella: To the Bandwidth Barrier and Beyond, by <http://dss.clip2.com>
- [2] The Gnutella Protocol Specification v0.4, by <http://dss.clip2.com>
- [3] GnutellaDev Online, <http://www.gnutelladev.wego.com>
- [4] NS Network Simulator, <http://www.isi.edu/nsnam/ns>