Measurement and Analysis of Gnutella Signaling Traffic

Dragos Ilie, David Erman, Adrian Popescu, Arne A. Nilsson

Dept. of Telecommunication Systems
School of Engineering
Blekinge Institute of Technology
371 79 Karlskrona, Sweden
{dragos.ilie, david.erman, adrian.popescu, arne.nilsson}@bth.se

***Abstract*— The paper reports on in-depth measurements and analysis of Gnutella signaling traffic collected at the Blekinge Institute of Technology (BIT), Karlskrona, Sweden. The measurements are based on a week-long packet trace collected with help of the well-known `tcpdump` application. Furthermore, a novel approach has been used to measure and analyze Gnutella signaling traffic. Associated with this, a dedicated `tcptrace` module has been developed and used to decode the packet trace, down to individual Gnutella messages. The measurement infrastructure consists of a Gnutella node running in ultrapeer mode and protocol decoding software. Detailed traffic characteristics have been collected and analyzed, such as session durations and interarrival times, and Gnutella message sizes and duration. Preliminary results show a high degree of variability of the Gnutella signaling traffic, which is mostly created by the `QUERY` messages. Furthermore, the Gnutella session interarrival times are observed to resemble the exponential distribution.**

***Index Terms*— P2P, Gnutella, traffic measurements, traffic characteristics, flow reassembly**

I. Introduction

The popularity of peer-to-peer (P2P) networking has lead to a dramatic increase of the volume and the complexity of traffic generated by P2P applications. Some of the most important reasons behind the popularity of P2P networking are related to the increased robustness through information redundancy (dissemination), ability to easily share own resources on demand, fine-grained access control policies, anonymity and encryption. Since the number of P2P nodes is still growing rapidly it is important to study the characteristics of P2P traffic, to assess the impact on network and to design control mechanisms to ensure that networking resources are fairly shared between P2P services and traditional services.

Well-known protocols like TCP and HTTP contain mechanisms for robust prevention and recovery to compensate for adverse networking conditions (e.g., TCP congestion control) and for optimal usage of network resources (e.g., HTTP persistent connections, range-request and expect/continue). They were designed and implemented at a time when Internet connections and computers were scarce and error-prone, and available to only a small number of people. Consequently, new protocols were developed with the aim to minimize the CPU power, to optimize link utilization and to improve the QoS. Today, the situation is very different. Large capacity, low error rate data links are being installed into residential areas and computer access is almost ubiquitous. The possibility to develop networking applications is available to everybody thanks to extensive software libraries. With the advent of Napster in the fall of 1999 [6], [15], the P2P revolution was started and, since then, the world has witnessed the rise and fall of many types of P2P protocols.

In contrast to the traditional networking services, P2P services provide data transfer without much consideration to fair and optimal use of network resources or side-effects on other network services [1]. They take instead a narrow view in the sense that they focus on where to find resources demanded by users and on how to maximize the throughput of individual data flows by selecting high-speed peers. On top of that, every node in a pure P2P network is typically involved into overlay routing, where decisions are taken based on specific constraints (e.g., connectivity, resource availability and expected data throughput) as to along which path to forward P2P data. The result is that the overlay routing is taking over some functionality generally handled by the network layer. Consequently, some of the assumptions considered at the transport layer about the services offered by the network layer may no longer be true in the case of P2P networking. To make things worse, the routing topology is in continuous change due to peers joining or leaving the overlay (the so-called "ad-hoc effect"). Peers depend on notifications from neighboring nodes in order to maintain up-to-date routing tables. Thus, the design choices for the overlay signaling and control as well as interactions with IP routing and transport protocols are important for the end-to-end performance. Furthermore, detailed traffic measurements are imperious to understand the Gnutella traffic patterns with the goal to find the optimal design for overlay signaling and control.

The paper reports on measurement and analysis of Gnutella signaling traffic collected at the Blekinge Institute of Technology (BIT), Karlskrona, Sweden. We have chosen Gnutella because of the openness of the protocol specification and the relative large user base. Gnutella uses a special packet format for peer signaling and resource discovery and a subset of HTTP for file transfers. The measurement infrastructure consists of a Gnutella node running in ultrapeer mode and protocol decoding software developed at the BIT. In Gnutella networks,

ultrapeers are nodes with better networking capabilities and higher CPU power that act as proxies on behalf of less capable nodes, the so-called leaf nodes. The beneficial side-effect of running an ultrapeer is the access to a large number of Gnutella peers.

Packet traces have been collected with the help of the well-known `tcpdump` application [12] and a significant volume of non-Gnutella traffic has been filtered out prior to decoding and analysis. A dedicated `tcptrace` module [16] has been developed to decode the Gnutella signaling traffic. Collected data include, among others, session duration, session interarrival times, Gnutella message sizes and duration. The objectives are to capture the type and sizes of protocol objects, to test for the presence of heavy-tailed properties, to find structural similarities or differences among characteristics of sessions as well as to search for possible invariant characteristics across object flows.

The motivation is to build stochastic models for key elements of the protocol that can be further used to build integrated performance testbed for conducting experiments on P2P traffic control and engineering.

The rest of the paper is organized as follows. In Section II the Gnutella protocol is described in detail. Section III describes the architecture of the measurement infrastructure for P2P developed at BIT. In Section IV we present the metrics measured, followed by the measurement results in Section V. Section VI concludes the paper.

II. The Gnutella Protocol

Gnutella is a decentralized P2P system [3], [13]. Participants can share any type of resources, although the currently available specification covers only file resources. The protocol is easily extensible, which has lead to the adoption of a variety of proprietary and non-proprietary extensions (e.g., ultrapeers [19] and the Query Routing Protocol [18]).

The activities of Gnutella peers can be divided in two main categories: *signaling* and *exchange of user resources* (further referred to as *resource or data exchange*).

The signaling activities are concerned with discovering the network topology (with the help of `PING` and `PONG` messages) and locating resources of interest (with the help of `QUERY` and `QUERY_HIT` messages). For example, a new Gnutella node will start with a list of potential peers to which will try to connect. For each successful connection, the connected peers will exchange the IP addresses of other peers they know about. Another example of signaling is when a peer searches for resources. The resource query is broadcasted to all directly connected peers. Each of these peers further forwards the query to all directly connected peers, except for the one where the message came from. The range of flooding is controlled by a Time-To-Live (TTL) parameter in the query header. Each peer that has the sought resources sends a reply message along the exact same path as the incoming query message. The reply message contains a description of all resources that match the query along with the socket address (i.e., IP address and port address) to use for downloading.

Resource exchange occurs when a peer has localized a resource of interest (e.g., a spreadsheet file or the ISO image for the latest Linux distribution). The peer attempts then to download the file by establishing a direct HTTP connection to the socket indicated in the query response message. The resource is finally downloaded by using the HTTP `GET` method. Both HTTP 1.0 [2] and HTTP 1.1 [8] must be supported by peers for data exchange and the later protocol is the preferred choice.

If the peer holding the resource is behind a firewall that blocks incoming connections, the peer that wants to download the resource can send a Gnutella `PUSH` message over the signaling channel. Upon receiving the `PUSH` message the firewalled host will attempt to open a TCP connection to the host requesting the `PUSH`. If successful, the firewalled host notifies its peer that it can initiate the HTTP download over the new connection. In the case both hosts are behind firewalls that block incoming connections, then data exchange is not possible.

A. Overlay topology hierarchy

Initially, Gnutella networks (referred to as Gnets in the rest of the paper) were non-hierarchical. However, it has been observed that the abundance of signaling was a major threat to the scalability of these networks [19]. Limewire (a company promoting an enhanced Gnutella *servent*[1]) suggested therefore the introduction of a two-level peer hierarchy: *ultrapeers* and *leaf* nodes. Ultrapeers are faster nodes in the sense that they are connected to high-capacity links and have a large amount of CPU power. Leaf nodes maintain a single connection to their ultrapeer. The typical ultrapeer maintains 10-100 connections, one for each leaf node, and 1-10 connections to other ultrapeers, again one connection for each ultrapeer node. The ultrapeers do signaling on behalf of the leaf nodes thus shielding leaf nodes from virtually all ping and query traffic [19]. A ultrapeer does not necessarily have leaf nodes, it can work standalone.

Some servents may not be capable to become leaf nodes or ultrapeers for various reasons (e.g., they lack the functionality). In this case, they are labeled *legacy* nodes. In order to improve the overall scalability of the Gnets and to preserve bandwidth, ultrapeers and leaf nodes may refuse to connect to legacy nodes.

B. Peer discovery

Peer discovery is done mainly through the use of *GWebCache* servers and `PING` – `PONG` messages.

A Gnutella node that wants to become a peer in an overlay must first have information about the listening socket for at least one peer that is already member in the overlay. This will be referred to as the *bootstrap problem*. The typical way to solve the bootstrap problem was to visit a web site that has recent lists of known peers on a web page. The next step was to select one of the peers listed on the page, cut-and-paste its address (i.e., the listening socket) from the web browser into the Gnutella servent and try to open a connection to it. This

[1]Servent denotes a software entity that acts either as a client or as a server. The name is a combination of the two terms: SERVer cliENT.

process would continue until at least one successful connection was made. At this point the PING – PONG traffic would, hopefully, reveal more peers to which the servent could connect. The addresses of newly found peers were cached locally and reused when the servent application was restarted. Since peers in general have a short life span (i.e., they enter and leave the network very often) the peer lists kept by each node often got outdated. GWebCache servers try to solve this problem. Each GWebCache server is essentially an HTTP server with a main web page that contains a list of listening sockets for known peers. The web page is typically rendered by a CGI script or Java servlet, which is also capable of updating the list contents. Ultrapeers update the list continuously ensuring that new peers can always find an overlay to join. Another list describing all available GWebCache servers is maintained at the main GWebCache web site. This list contains only GWebCache servers that have elected to register themselves. Unofficial GWebCache servers exist as well, providing some sort of anonymity to peers using them.

The new way to bootstrap a Gnutella node is to have it connect to the GWebCache web site, obtain the list of GWebCache servers, try to connect to a number of them, and finally build up a list of active servents. Alternatively, the node can connect to an unofficial GWebCache server or connect directly to a node in the Gnet.

C. Connection establishment

Assuming a Gnutella servent has obtained the listening socket of a peer, it can attempt to open a signaling channel by establishing a full-duplex TCP connection to the specific peer. In the following, the servent that has done the TCP active open will be referred to as the *client*, and its peer will be referred to as the *server*. Once the TCP connection is in place a handshaking procedure is done between the client and the server[2]:

1) The client sends the string GNUTELLA CONNECT/0.6<CR><LF> where <CR> is the ASCII code for carriage return and <LF> is the ASCII code for line feed.
2) The client sends all capability headers in a format similar to HTTP and ends with <CR><LF> on an empty line, e.g.,
   ```
   User-Agent: BearShare/1.0<CR><LF>
   X-Ultrapeer: True<CR><LF>
   Pong-Caching: 0.1<CR><LF>
   <CR><LF>
   ```
3) The server responds with the string GNUTELLA/0.6 <status-code><status-string><CR><LF>. The <status-code> follows the HTTP specification with code 200 meaning success. The <status-string> is a short human readable description of the status code (e.g., when the code is 200 the string will typically be set to OK).
4) The server sends all capability headers as described in step 2.
5) The client parses the server response to compute the smallest set of common capabilities available. If the client still wishes to connect, it will send GNUTELLA/0.6 <status-code> <status-string><CR><LF> to the server with the <status-code> set to 200. If the capabilities do not match, the client will set the <status-code> to an error code and close the TCP connection.

If the handshake is successful, the client and the server start exchanging binary Gnutella messages over the existing TCP connection. The existing TCP connection lasts until one of the peers decides to terminate the session. At that point the peer ending the connection has the opportunity to send an optional Gnutella BYE message. Then the peer closes the TCP connection.

If the capability set used by the peers includes stream compression then all data on the TCP connection, with the exception of the initial handshake, will be compressed [14]. The type of compression algorithm can be selected in the capability header, but the *de-facto* standard seems to be *deflate*, which is implemented in zlib [10].

Each Gnutella message starts with a generic header that contains the following fields:

- Message ID/GUID (Globally Unique ID) to uniquely identify messages on Gnet. The GUID is a combination of the peer's MAC address and a timestamp.
- Payload type code that identifies the type of Gnutella message (e.g., PONG messages have payload type 0x01).
- TTL (Time-To-Live) to limit the signaling radius and the adverse impact on the network. Messages with TTL greater than 15 should be dropped according to the protocol specification.
- Hop count to inform receiving peers how far the message has traveled (in number of peer hops).
- Payload length to describe the total length of the message following the header. The next generic message header is located exactly this number of bytes from the end of this specific header. Since there is no specific framing that separates messages apart the servent must use the size field to discover message boundaries.
- The generic Gnutella header is followed by the actual message that may have own headers.

D. Peer exchange

Every successfully connected pair of peers starts sending periodic PING messages one to the other. The receiver of the PING message decrements the TTL in the message header and increments the Hops field. If the TTL is not zero, then the message is forwarded to all directly connected peers (with the exception of the one from where the message came). PING messages do not carry any information (not even the sender listening socket), which means that the payload length in the Gnutella header is zero.

PONG messages are sent only in response to PING messages. More than one PONG message can be sent in response to

[2]The handshaking procedure uses elements that are similar to HTTP. It is important to point out that this format was selected out of convenience. The signaling traffic is not pure HTTP.

a PING. The PONG messages are returned on the reverse path used by the corresponding PING message. Each PONG message contains detailed information about one active Gnutella peer along with the GUID from the PING message that triggered it. The PONG receiver can then attempt to connect to the peer that was described in the message.

Ultrapeers use a similar procedure, however they do not forward PING and PONGs to/from the leaf nodes attached to them.

E. Query signaling

Gnutella peers can locate resources through the use of QUERY messages sent to directly connected peers over the TCP connection established during the handshake. A QUERY message specifies in plain text the name of a resource and the minimum speed (i.e., link capacity) expected from the servents that respond to this message. There may also be additional message extensions immediately following standard QUERY message data (e.g., proprietary extensions), these are however not considered in our study. Peers receiving a QUERY message forward it to all directly connected ultrapeers unless the TTL field indicates otherwise.

If a peer that has received the QUERY message is able to serve the resource, it responds with a QUERY_HIT message. The GUID for the QUERY_HIT message must be the same as the one in the QUERY message that has triggered the response. The QUERY_HIT message lists each resource name that matches the resource query. For example, the string linux could identify a resource called linux_redhat_7.0.iso as well as a resource called linux_installation_guide.txt.gz. Thus, this fictive query can be answered with a QUERY_HIT message containing two results along with the size in bytes of each resource. In addition, the QUERY_HIT messages contain information about the listening socket to be used by the message receiver when it wants to download the resource. The protocol specification limits the total size of a QUERY_HIT message. Consequently, several QUERY_HIT messages can be issued by the same servent in response to a QUERY message.

When the QUERY_HIT receiver decides to download the resource described in the message it will first try to establish a direct HTTP connection to the listening socket obtained. If the QUERY_HIT sender (i.e., the resource owner) is behind a firewall, then the downloader will send a PUSH message to it using the reverse path of the QUERY_HIT message. Using the PUSH message the resource owner can establish a TCP connection to the downloader. The downloader can then use the HTTP GET method to retrieve the resource over the new connection.

F. Ultrapeers and QRP

The mission of ultrapeers is to reduce the burden put on the network by peer signaling. They achieve this goal by eliminating the PING messages among leaf nodes and by employing query routing. There are various schemes for ultrapeer query routing but the recommended one is the Query Routing Protocol (QRP) [18]. Ultrapeers signal among themselves by using PING and PONG messages.

The Query Routing Protocol (QRP) was introduced in order to mitigate the adverse effects of the broadcasts used for Gnutella queries and it is based on a modified version of Bloom filters [11]. The idea is to decompose a query text string into individual keywords and have a hash function that is applied to each keyword in the query. Given a keyword, the hash function returns an index to an element in a finite discrete vector. Each entry in the vector is the minimum distance expressed in number of peer hops to the peer holding a resource that matches the keyword in the query. Queries are forwarded only to hosts that have resources that match all keywords. By this, the bandwith used by queries is substantially reduced. Peers run the hash algorithm over the resources they share and exchange the routing tables (i.e., hop vectors) at regular intervals.

Individual peers (legacy or ultrapeer nodes) may run QRP and exchange routing tables among themselves [9]. However, the typical scenario is that legacy nodes do not use QRP, leaf nodes send route table updates to ultrapeers only, and ultrapeers propagate these tables only to directly connected ultrapeers.

G. Miscellaneous signaling

There are several miscellaneous messages flowing over Gnets such as PUSH and BYE messages or other messages based on proprietary Gnutella extensions.

PUSH messages are used by peers that want to download resources from peers located behind a firewall, which prevents incoming TCP connections. The downloader sends a PUSH message over the existing TCP connections, which was setup during the handshake phase. The PUSH message contains the listening socket of the downloader. The host behind the firewall can then attempt to establish a TCP connection to the listening socket described in the message. If the TCP connection is established successfully, the firewalled host sends the following string over the signaling connection:

```
GIV <File Index>:<Servent Identifier> \
/<File Name><CR><LF>
```

The backslash (\) character at the end of the line indicates an artificial line break for the sake of page formatting (i.e., in reality GIV is sent as one line). The <File Index> and <Servent Identifier> are the values found in the PUSH message received previously and <File Name> is the name of the resource requested. Upon receiving this message, the non-firewalled host issues a HTTP GET request over the newly established TCP connection:

```
GET /get/<File Index>/<File Name> \
HTTP/1.1<CR><LF>
User-Agent: Gnutella<CR><LF>
Connection: Keep-Alive
Range: bytes=0-<CR><LF>
<CR><LF>
```

The BYE message is an optional message used by servents to inform the directly connected peers that the signaling

connection will be closed. The message contains an error code along with an error string. The message is sent only to hosts that have indicated during handshake that they support this message type.

H. Data transfer

Data exchange is done over a direct HTTP connection between a pair of peers. Both HTTP 1.0 and HTTP 1.1 are supported and the later is recommended. Most notably, the persistent connection and range request features are preferential. The persistent connection allows a pair of peers to transfer several files over the same HTTP connection.

The range request allows a peer to continue an unfinished transfer from where it left off. Furthermore, it allows servents to utilize *swarming*, which is the technique to retrieve different parts of the file from different peers. This allows more efficient use of the bandwidth. Swarming is not part of the Gnutella protocol, regular Gnutella servents (i.e., servents that do not explicitly support swarming) can nevertheless engage in swarming without being aware of it. From their point of view, a peer is requesting a range of bytes for a particular resource. The intelligence resides with the peer downloading the data.

Fig. 1 shows a simple Gnet scenario, involving three legacy peers. It is assumed that Peer A has obtained the listening socket of Peer B from a GWebCache server. Using the socket descriptor, Peer A attempts to connect to Peer B. In this example, Peer B already has a signaling connection to Peer C.

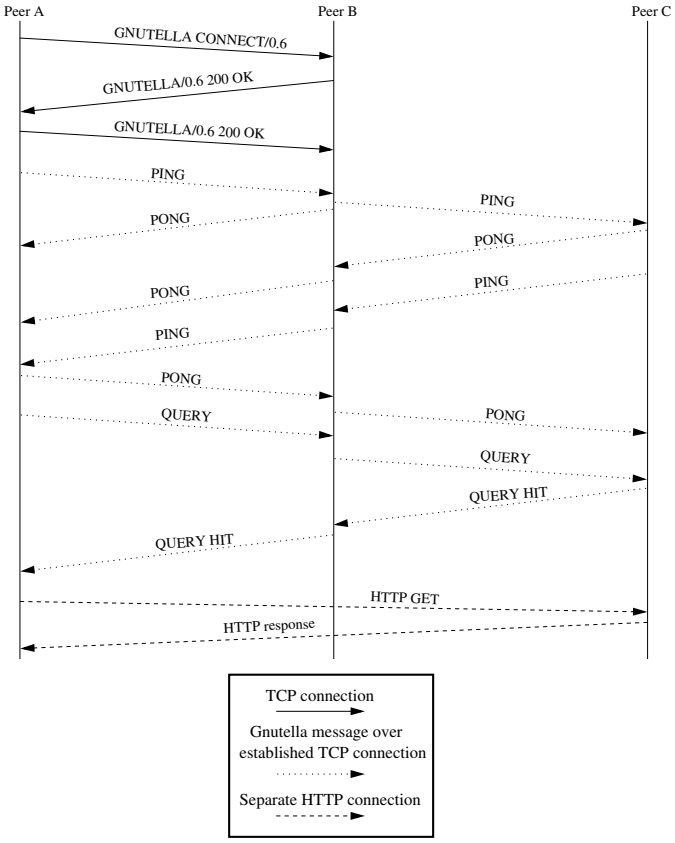

Fig. 1. Example of Gnutella session

The first three messages between Peer A and Peer B illustrate the establishment of the signaling channel between the two peers. The two peers may exchange capabilities during this phase as well.

The next phase encompasses the exchange of network topology information with the help of `PING` and `PONG` messages. The messages are sent over the TCP connection established previously (i.e., during the peer handshake). It is observed that `PING` messages are forwarded by Peer B from Peer A to Peer C and in the opposite direction as well as that `PONG` messages follow the reverse path taken by the corresponding `PING` message.

At a later time the Peer A sends a `QUERY` message, which is forwarded by Peer B to Peer C. In this example, only Peer C is able to serve the resource, which is illustrated by the `QUERY_HIT` message. The `QUERY` and `QUERY_HIT` messages use the existing TCP connection, just like the `PING` and `PONG` messages. Again, it is observed that the `QUERY_HIT` message follows the reverse path taken by the corresponding `QUERY` message.

Finally, Peer A opens a direct HTTP connection to Peer C and downloads the resource by using the HTTP `GET` method. The resource contents are returned in the HTTP response message.

The exchange of `PING` - `PONG` and `QUERY` - `QUERY_HIT` messages continues until one of the peers tears down the TCP connection. A Gnutella `BYE` message may be sent as notification that the signaling channel will be closed.

III. Measurement Infrastructure

The P2P measurement infrastructure developed at BIT [7] consists of a ultrapeer node and protocol decoding software. Further, the protocol decoding software is based on two well-known applications: `tcpdump` [12] and `tcptrace` [16]. Although the infrastructure is currently geared towards P2P protocols, it can be easily extended to measure other protocols running over TCP [7] as well. Furthermore, we plan also to develop similar modules to measure UDP-based applications.

The measurement procedure can be best described as a 5-stage process, as illustrated in Fig. 2. The stages are: data collection, TCP reassembly, application message flow reassembly, log data reduction and postprocessing/analysis. Only the first stage deals with live data. The other four stages process data off-line.

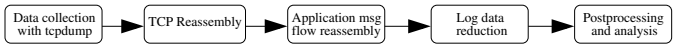

Fig. 2. Measurement procedure

A. Data collection

The data collection stage used at BIT consists of a Gnutella node running in ultrapeer mode. The benefit of running an ultrapeer node for the purpose of measurements is that the node will see more traffic. The reason is two-fold: first, the node acts as an application layer router between leaf nodes and other ultrapeers, leaf and legacy nodes and, secondly, most

nodes prefer to connect to ultrapeers since ultrapeers provide better connectivity than leaf and legacy nodes.

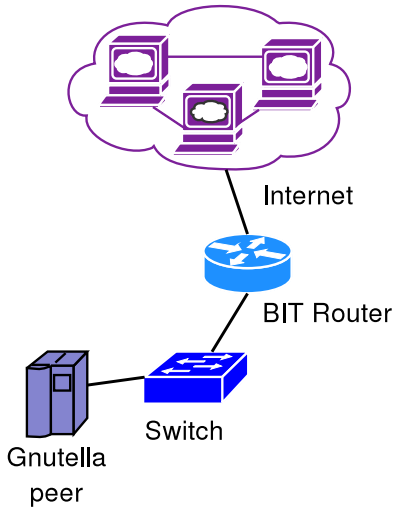

Fig. 3. Measurement setup

The BIT ultrapeer node runs the Gentoo Linux 1.4 operating system, with kernel version 2.6.5 and the `gtk-gnutella-0.93.3` servent software. The node is equipped with an Intel Celeron 2.4GHz processor, 1GB RAM, 120GB hard drive, and 10/100 FastEthernet network interface. As shown in Fig. 3, the network interface is connected to a 100Mbit switch in the lab at our department, which is further connected through a router to the SUNET backbone. In addition, the node has `tcpdump 3.8.3` installed on it. When the node is running measurements, `tcpdump` is started before the Gnutella servent in order to avoid missing any connections. `Tcpdump` can also be run on a different node in the network, provided that the ultrapeer switch port is mirrored to the port where the `tcpdump` host is recording or if the switch is replaced with a hub and both the `tcpdump` host and the ultrapeer are connected to it.

During the data collection stage, `tcpdump` collects Ethernet frames from the switch port where the ultrapeer node is connected. Since most P2P applications can use dynamic ports, all traffic reaching the switch port must be collected. However, to increase the performance during data collection and data processing, one can turn off most or all server software on the ultrapeer node. It is possible, in addition, to apply a filter to `tcpdump` that drops packets used by traditional services, which are running on well-known ports (e.g., HTTP, FTP, SSH).

The volume of collected data can be quite large, e.g., the resulting trace file could grow well beyond 4 GBytes in less than one day, which is larger than most standard filesystems can handle without modification. This is directly related to the number of peers the servent is allowed to connect to. In our case we had on average 150 peers (100 leaf nodes and 50 ultrapeers) and collected approximately 75 GBytes pcap data in seven days. The solution was to have `tcpdump` spread the recorded data across several files, each 600 MBytes large. The file size was set to 600 MBytes to have each data file small enough to fit on a recordable CD.

B. TCP Reassembly

Assuming that the measured P2P application runs over TCP, the next step is to reassemble the TCP frames to a flow of ordered bytes. The TCP reassembly module builds on the TCP engine available in `tcptrace` [7].

The module reads the `tcpdump` traces in the order they were created. Each trace is scanned for TCP connections. When found, they are stored in a list with connection records. Further, when a new TCP segment is found in the trace file, the module scans the connection list comparing the socket pair of the segment with each entry in the list. If no entry matches the socket pair of the new segment, then a new connection is considered to be found and a record is created for it, which finally is added to the connection list. Otherwise, the connection record matching the socket pair is retrieved and sent together with the new segment to the TCP reassembly engine.

The TCP reassembly engine is similar to the one used by the FreeBSD TCP/IP stack as described in [17]. For each active connection, the reassembly engine keeps a doubly linked list, which is referred to as the reassembly list. When given a connection record and a new segment, it retrieves the correct reassembly list and then it inserts the new segment in the correct place in the list. The reassembly engine is capable of handling out-of-order segments as well as forward and backward overlapping between segments.

C. Application data flow reassembly

Whenever new data is available, the application data reassembly module is notified. Upon notification, it will ask the TCP reassembly module for a new segment from the reassembly list corresponding to the socket pair received with the notification. When it receives the new segment, it interprets the contents according to the specification for the protocol it decodes. Since application messages may span several segments and since a segment may contain data from two consecutive messages, each segment is appended to the end of a data buffer before further processing, thus creating a contiguous data flow containing at least one application message.

In the case of a new Gnutella connection, the application reassembly module first waits for the handshake phase to begin. If the handshake fails the connection is marked invalid and it is eventually discarded by the memory manager.

If the handshake is successful, the application reassembly module scans the capability lists sent by the nodes involved in the TCP connection. If the nodes have agreed to compress the data, the connection is marked as compressed. Further segments received from the TCP reassembly module for this connection are first sent to the decompressor, before being appended to the data buffer.

The decompressor uses `zlib`'s [10] *inflate()* function to decompress the data available in the new segment. Upon

successful decompression the decompressed data is appended to the data buffer.

Immediately after the handshake phase, the application reassembly module attempts to find the Gnutella message header of the first message. Using the payload length field, it is able to discover the beginning of the following message. This is the only way to discover message boundaries and it is essential in order to be able to follow the flow. Based on the message type field in the message header, the corresponding decoding function is called, which outputs a message record to the log file. The message records follow a specific format required by the postprocessing stage.

D. Data reduction and postprocessing

Since the logs can grow quite large, they can be processed through an optional stage of data reduction. Data reduction is achieved by using the on-the-fly `gzip` compression offered by `zlib` [10]. Additional data reduction can be achieved if the user is willing to sacrifice some detail by aggregating data over time.

The postprocessing module interprets the (optionally compressed) log data and it is able to demultiplex it based on different types of constraints: message type, IP address, port number, etc. The data output format of this stage is suitable for input to numerical computation software such as MATLAB and standard UNIX text processing software such as `sed`, `awk` and `perl`.

IV. Traffic Metrics

The measurement infrastructure developed at BIT offers the possibility to extract traffic metrics for every layer in the TCP/IP stack [7]. In the following, we report on application layer metrics only, with emphasis on metrics used to characterize the Gnutella signaling traffic. These metrics can be partitioned into two main classes, namely *session metrics* and *message metrics*.

A. Session metrics

A Gnutella session is defined to be the collection of events resulting from the interaction between two Gnutella peers that communicate with each other. A session begins with a handshake message and it is terminated by either a `BYE` message or by tearing down the corresponding TCP connection (Fig. 1).

A number of metrics have been used for the characterization of Gnutella signaling traffic [7]. The most important session metrics are as follows:

Session interarrival time: this is the time duration between the initial handshake messages of two consecutive sessions. We differentiate between *incoming* sessions and *outgoing* sessions. Outgoing sessions are sessions where the initial handshake message is sent by the ultrapeer at BIT and the rest of sessions are incoming sessions.

Session duration: this is the time duration for a session from the moment when a handshake is attempted until the time of the last Gnutella message on the specific connection.

Number of messages in a session: this is the number of messages exchanged during a session between the peers involved in the specific session.

Number of bytes in a session: this is the number of bytes transferred in messages exchanged during a session between the peers involved in the specific session. This metric does not include the bytes corresponding to the link layer, IP and TCP headers.

B. Message metrics

A Gnutella message consists of the generic Gnutella header and message data. The message data may contain additional headers and third-party extensions.

The most important message metrics are as follows:

Message type: this refers to the type of Gnutella message as extracted from the type field in the Gnutella header.

Message size: this refers to the size of the message as obtained from the length field in the Gnutella header.

Message duration: this refers to the time needed to receive all TCP segments belonging to the specific message. A message that spans one TCP segment or a fraction of a TCP segment is considered to have zero duration.

Message rate: this metric refers to the number of messages per second for the specific message type.

Message byte rate: this metric refers to the number of message bytes (including the Gnutella header) per second for the specific message type.

V. Results

The reported Gnutella signaling traffic is based on a week-long packet trace collected at BIT. The recorded traffic volume sums up to more than 282 million IP datagrams, which were used to carry 763 million Gnutella messages in more than 773 thousand Gnutella sessions. Out of these, only about 16 thousand sessions successfully completed the handshake phase, as it is observed in Table II.

TABLE I
SERVER HANDSHAKE CODE

Response code	Code meaning	Count
200	OK	27357
401	Unauthorized	1
403	Gnet connection not compressed	203131
404	Already connected	3198
406	Protocol not acceptable	3091
503	Full	535479
550	Hostile IP address banned	1085

TABLE II
FINAL HANDSHAKE CODE

Response code	Code meaning	Count
200	OK	16039
401	Unauthorized	1
403	Gnet connection not compressed	3320
503	I am a shielded leaf node	5873
577	Service not available	101

Table I reports a summary of the number of various response codes that were sent during the server-side part of the handshake (Fig. 1). Similarly, Table II shows the number of various response codes that were sent during the final phase of the handshake.

TABLE III
MESSAGE TYPE DISTRIBUTION

Message type	Number of messages	Percentage of total
CLIHSK	777040	0.101
SERHSK	773342	0.101
FINHSK	25333	0.003
PING	5797304	0.760
PONG	34353576	4.503
QUERY	677458351	88.800
QUERY_HIT	42557130	5.578
QRP	25478	0.003
VEND	63817	0.008
STD_VEND	0	0.000
PUSH	1055452	0.138
BYE	11328	0.001
UNKNOWN	4050	0.001
Total	762902201	100.000

Table III shows a summary of the number of messages of different types found in the data set. It is observed that QUERY messages are responsible for most (88.8%) of the traffic volume, followed by QUERY_HIT (about 5.6%) and PONG messages (4.5%). CLIHSK, SERHSK and FINHSK represent messages sent during the Gnutella connection establishment procedure. CLIHSK is the initial client handshake, followed by the server handshake SERHSK and finally by the final part of the handshake FINHSK. The UNKNOWN message field accounts for Gnutella messages that use an unknown message type value. We finally observe the absence of any standard vendor STD_VEND messages. We hypothesize that this is because of a combination of lack of standardization procedure and vendor disinterest.

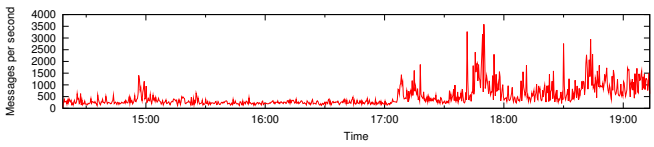

Fig. 4. Overall message rate of the BIT Gnutella node

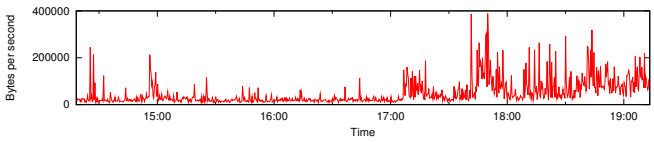

Fig. 5. Overall byte rate of the BIT Gnutella node

Fig. 4 and Fig. 5 show the overall message rate and overall byte rate respectively, which were measured at the BIT Gnutella node immediately after it was started. These metrics refer to the incoming traffic plus the outgoing traffic. These are results that confirm the transition from leaf node to ultrapeer of the BIT Gnutella node. One of the criteria used to decide if a node is eligible to become ultrapeer is long up-times, in the order of several hours [13], [19]. Both figures show that the overall traffic volume, expressed in number of messages per second and number of bytes per second respectively, increases suddenly after the node was running for four hours. This is a sign that the node was recognized by other peers as an ultrapeer.

Fig. 6 and Fig. 7 show the number of incoming and outgoing messages per second measured at the BIT Gnutella node. It is observed that the outgoing message rate is larger, on the average, than the incoming message rate. Table IV summarizes the main statistics associated with these metrics.

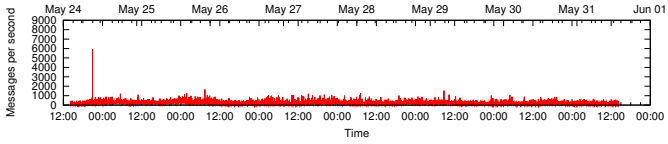

Fig. 6. Incoming message rate at the BIT Gnutella node

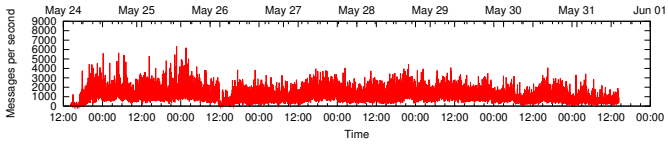

Fig. 7. Outgoing message rate at the BIT Gnutella node

TABLE IV
MESSAGE RATE STATISTICS

Dir	Samples	Max	Min	Mean	Variance	Stddev
IN	597026	86369	1	215	34980	187
OUT	597026	8182	1	1063	346057	588

Fig. 8 and Fig. 9 give an alternative view of the traffic volume expressed as incoming byte rate and outgoing byte rate. The number of bytes covers the Gnutella header and the Gnutella payload, but not the link-layer and TCP/IP headers. It is mentioned that Fig. 8 truncates some of the spikes, which will otherwise dwarf the majority of the traffic volume. Preliminary investigations indicate that these spikes occur due to the combined effect of long TCP reassembly queues in the TCP/IP stack and exceptionally good compression ratios with *zlib* [4], [5], [10].

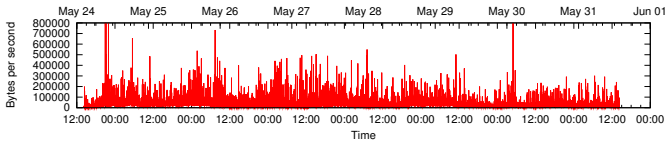

Fig. 8. Incoming byte rate measured at the BIT Gnutella node

Table V summarizes the associated statistics for the incoming and outgoing byte rates. It is observed the high variance in the number of bytes per second for both incoming and outgoing traffic. The consequence is that we expect the Gnutella signaling traffic to have a high degree of heavy-tailedness.

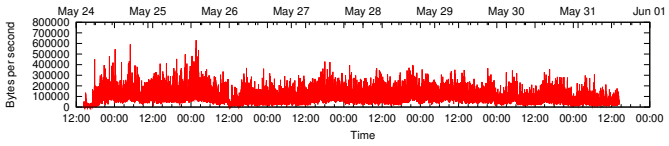

Fig. 9. Outgoing byte rate measured at the BIT Gnutella node

Further studies on traffic self-similarity will however be done in the future.

TABLE V
MESSAGE BYTE RATE STATISTICS

Dir	Samples	Max	Min	Mean	Variance	Stddev
IN	597026	49863052	23	31280	10134416588	100669
OUT	597026	738185	21	93884	2985683010	54641

Fig. 10 through Fig.13 show examples of other metrics (PING and PONG message rates) captured with the measurement infrastructure. It is observed that the volume of incoming PING traffic is much larger than the volume of the outgoing PING traffic, whereas an opposite relation is observed for the PONG traffic.

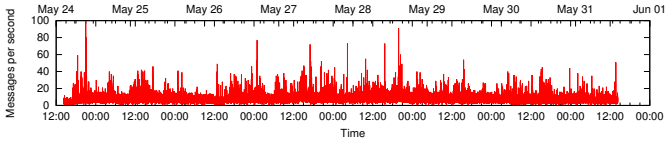

Fig. 10. Incoming PING message rate

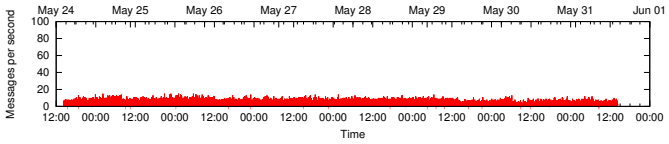

Fig. 11. Outgoing PING message rate

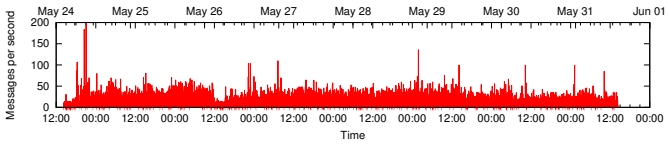

Fig. 12. Incoming PONG message rate

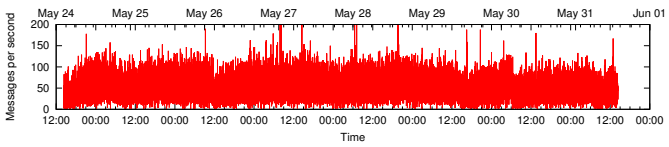

Fig. 13. Outgoing PONG message rate

Fig. 14 and Fig. 15 show a huge disproportion in the number of incoming QUERY messages as compared to the number of outgoing QUERY messages. Table VI summarizes the essential statistics. Further investigation needs to be done in order to explain this large skew in the proportion between incoming and outgoing QUERY messages, and this is subject for future work.

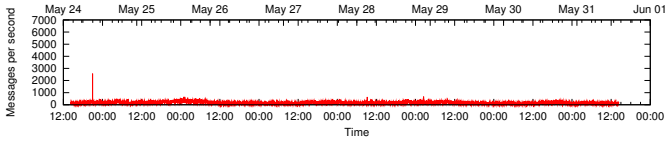

Fig. 14. Incoming QUERY message rate

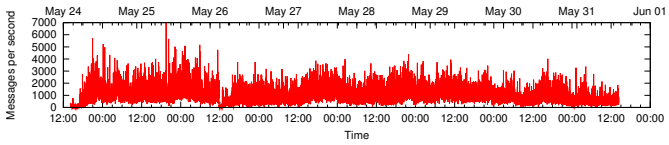

Fig. 15. Outgoing QUERY message rate

TABLE VI
QUERY MESSAGE RATE STATISTICS

Dir	Samples	Max	Min	Mean	Variance	Stddev
IN	597019	52022	1	160	12123	110
OUT	596906	8021	1	975	331937	576

Finally, Fig. 16 shows an example of histogram that we have observed for the session interarrival times. It is observed that the shape of the histogram resembles the exponential distribution.

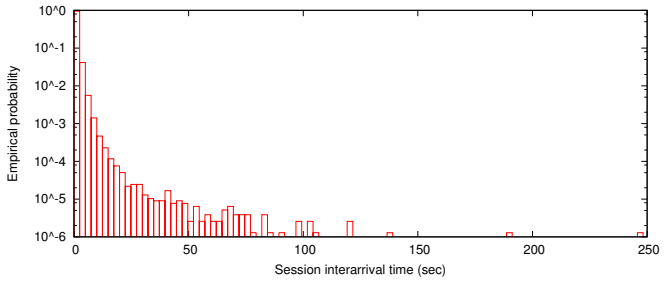

Fig. 16. Empirical distribution for the session interarrival time

VI. CONCLUSIONS

A measurement study of Gnutella signaling traffic has been reported for traffic collected at the Blekinge Institute of Technology (BIT), Karlskrona, Sweden. A novel approach has been used for traffic measurements that is based on combining TCP stream identification and extraction with application data flow reassembly. Its main advantage lies in the information detail it provides, which is down to individual fields in the Gnutella message headers. The measurement infrastructure consists of a Gnutella node running in ultrapeer mode, with the benefit of access to a large number of Gnutella peers. Preliminary results show a high degree of variability of the Gnutella signaling traffic, which is mostly created by QUERY messages. Furthermore, the Gnutella session interarrival times are observed to resemble the exponential distribution.

The data obtained from the measurements could be used to characterize a Gnutella ultrapeer under high load. Our plans for future work include workload characterization of ultrapeers and leaf nodes under low to medium load. The different types of workload characterization will enable us to simulate different strategies used to improve the overlay performance.

In addition to that, future work will be about further analysis of the obtained results, to find out structural similarities or differences among characteristics of Gnutella ultrapeers and leaf nodes as well as to search for possible invariant characteristics across object flows.

REFERENCES

[1] Eytan Adar and Bernardo A. Huberman. Free riding on gnutella. *First Monday*, 5(10), October 2000. http://firstmonday.org/issues/issue5_10/adar/index.html.

[2] T. Berners-Lee, R. Fielding, and H. Frystyk. *Hypertext Transfer Protocol – HTTP/1.0*, May 1996. RFC 1945.

[3] Clip2. *The Annotated Gnutella Protocol Specification v0.4*. The Gnutella Developer Forum (GDF), 1.8th edition, July 2003. http://groups.yahoo.com/group/the_gdf/files/Development/.

[4] P. Deutsch. *DEFLATE Compressed Data Format Specification version 1.3*, May 1996. RFC 1951.

[5] P. Deutsch and J-L. Gailly. *ZLIB Compressed Data Format Specification version 3.3*, May 1996. RFC 1950.

[6] Wikipedia Encyclopedia. Napster. http://en.wikipedia.org/wiki/Napster.

[7] D. Erman, D. Ilie, and A. Popescu. Peer-to-peer traffic measurements. Technical report, Blekinge Institute of Technology, Karlskrona, Sweden, 2004.

[8] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. *Hypertext Transfer Protocol – HTTP/1.1*, June 1999. RFC 2616.

[9] A. Fisk. *Gnutella Ultrapeer Query Routing*. Lime Wire LLC, 0.1 edition, May 2003. http://groups.yahoo.com/group/the_gdf/files/Proposals/Working Proposals/search/Ultrapeer QRP/.

[10] Jean-loup Gailly and Mark Adler. zlib. http://www.gzip.org/zlib.

[11] Burton H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Communication of the ACM*, Volume 13(Number 7):p. 422–426, July 1970. ISSN:0001-0782.

[12] Van Jacobsen, Leres C., and McCanne S. Tcpdump. http://www.tcpdump.org.

[13] Tor Klingberg and Raphael Manfredi. *Gnutella 0.6*. The Gnutella Developer Forum (GDF), 200206-draft edition, June 2002. http://groups.yahoo.com/group/the_gdf/files/Development/.

[14] Raphael Manfredi. *Gnutella Traffic Compression*. The Gnutella Developer Forum (GDF), January 2003. http://groups.yahoo.com/group/the_gdf/files/Development/.

[15] Napster. Napster. http://www.napster.com.

[16] Shawn Ostermann. Tcptrace. http://www.tcptrace.org.

[17] Gary R. Wright and W. Richard Stevens. *TCP/IP Illustrated: The Implementation*, volume 2. Addison-Wesley, 1995. ISBN: 0-201-63354-X.

[18] Christopher Rohrs. *Query Routing for the Gnutella Network*. Lime Wire LLC, 1.0 edition, May 2002. http://groups.yahoo.com/group/the_gdf/files/Development/.

[19] Anurag Singla and Christopher Rohrs. *Ultrapeers: Another Step Towards Gnutella Scalability*. Lime Wire LLC, 1.0 edition, November 2002. http://groups.yahoo.com/group/the_gdf/files/Development/.