

Implementation of a P2P system within a network simulation framework

Vinay Aggarwal

Anja Feldmann

Sebastian Mohr

Technische Universität München, Germany
{vinay,anja,mohrs}@net.in.tum.de

Abstract

To enable extensive experimentation with peer-to-peer (P2P) systems in a controlled environment, we have implemented a P2P system in a network simulation framework. In this paper, we explain the implementation process and the design issues we faced.

1 Motivation

In the recent past, peer-to-peer (P2P) systems have generated a lot of interest in the research community. Much of the attention has been focused on characterizing P2P systems, by capturing real traffic and subjecting it to analysis [6]. We intend to experiment extensively with P2P systems in a controlled environment, so that we can optimize our experiments to better suit our goals. To achieve this capability, we have implemented a P2P system in a network simulation framework.¹ The P2P protocol we chose to experiment with is Gnutella, and we coded it in the SSFNet simulation framework.

The reason behind choosing Gnutella is that it is an open-source protocol. It is also one of the most popular P2P systems in use. According to Slyck [2], it is the third most widely used file-sharing system, with around 1.6 million users world-wide. Besides, due to the massive interest that it has generated in the research community, there are some statistics available on the real-world behaviour and characteristics of Gnutella [11, 12]. This should help us later in validating our simulation model, when we monitor the application performance of our simulation framework by comparing our generated statistics with those already available for real-world Gnutella behaviour.

We begin by introducing the Gnutella protocol, followed by a brief description of SSFNet. We then proceed to give a detailed description of how we coded the Gnutella protocol in SSFNet. Here we explain at each stage, the challenges we faced, and

the various design decisions we made. Later, we describe how we correlate the P2P structure to simulated topology. This is followed by a discussion on the ongoing work of monitoring the application performance to reflect real-world behaviour. After giving some directions for future work, we conclude with a summary.

2 Introduction to Gnutella

One of the first decentralized overlay systems, Gnutella gives its participants the ability to share and locate resources hosted by other members of the network. It is a P2P system in the sense that, there is no distinction of members into clients and servers, rather, all members are equal and can initiate as well as serve requests. In other words, the same member can act as a client as well as a server in different circumstances.

A participant of Gnutella network, called a *servent*, is a computer system running an implementation of the Gnutella protocol. When launched, a servent searches for other servents in the Gnutella network, to whom it can connect. Each servent may or may not share any resources, and can search for desired resources within the network. While the general notion of resources tends to be music files, they can actually be anything from mapping to other resources, cryptographic keys, files of any type, to meta-information on keyable resources [1].

The servents interact with each other to share information on the resources that they offer, to query for desired resources, and to obtain responses to their queries. Based on the results, a servent decides which resource to obtain from which servent, and then, initiates the actual download of the resource.

While the negotiation traffic is carried within the set of connected Gnutella nodes, the actual data exchange of resources takes place outside the Gnutella network, using the HTTP protocol. In other words, the Gnutella network is only used to locate the nodes sharing the desired resources.

The messages used in the Gnutella protocol are

¹This work was partially supported by European Commission - FET Open Project DELIS - Dynamically Evolving Large Scale Information Systems - under the Complex Systems initiative

as follows [1].

1. **Ping:** It is a simple Hello-like message sent by a servent to actively discover other hosts on the network. It is also used to declare its own presence in the network.
2. **Pong:** It is a response to the **Ping**. A servent includes its own address and some information about the data (like number of files, etc.) that it shares on the network.
3. **Query:** This message is used to search the Gnutella network for desired resources. It is something like a simple question - "Does anybody have this file?"
4. **QueryHit:** It is a response to a **Query**. A servent possessing the requested resource replies with some information about its network connectivity (speed, etc.) to allow the questioner to suitably choose which node to download the resource from.
5. **Push:** A special message to allow a firewalled servent to share data.

3 Instrumentation in a simulation framework

To prepare a Gnutella simulation framework, we decided to extend the SSFNet simulation framework with the Gnutella protocol.

3.1 Introduction to SSF

The **Scalable Simulation Framework** (SSF) is an open-source standard for simulating large and complex networks. Written in Java, it supports discrete-event simulations [3].

SSF Network Models (SSFNet) are Java models of different network entities, built to achieve realistic multi-protocol, multi-domain Internet modeling and simulation. These entities include Internet protocols like IP, TCP, UDP, BGP4, and OSPF, network elements like hosts, routers, links, and LANs, and their various support classes.

Domain Modeling Language (DML) is a public-domain standard for model configuration and attribute specification. It supports extensibility, inheritance and substitution of attributes.

SSFNet is a collection of Java SSF-based components for modeling and simulation of Internet protocols and networks at and above the IP packet level of detail. Link layer and physical layer modeling can be provided in separate components.

SSFNet models are self-configuring - that is, each SSFNet class instance can autonomously configure

and instantiate itself by querying network configuration files written in the DML format. The principal classes used to construct Internet models are organized into two frameworks, SSF.OS (for modeling the host and operating system components, eg. protocols) and SSF.Net (for modeling network connectivity, and configuring nodes and links).

A more comprehensive and detailed explanation of the various facets of SSF can be found at [3].

However, what is noteworthy here is that the SSFNet provides us the framework on which we build our Gnutella model. Put another way, SSF provides the coding for the lower layers of IP stack, on which we *weave* the Gnutella protocol specification at the application layer. Naturally, a challenging aspect of the job is to fit the Gnutella code onto the SSF code, more specifically, the interaction of the two systems at the TCP layer. An explanation of this follows in the next section.

4 Coding for Gnutella

Now we arrive at the crux of this work, the actual implementation of Gnutella in SSFNet. We followed the Request for Comments (RFC) for Gnutella version 0.6 [1] for preparing the Gnutella model. In brief, we took the RFC information and converted it into code step by step, taking care at each step to carefully place the code on top of the SSF infrastructure. This led to many interesting, and at times, challenging propositions. For e.g., in the original SSF code, a node needs to tell the other communicating node, the exact size of the object being transferred. This added unnecessary complication to our proposed Gnutella model. Hence, we made changes to the SSFNet socket implementation so that it can self-compute the size of objects being transferred through them. Also, there was no queuing support built into the sockets, i.e., one could not write any data into a socket unless it was free. So we added support for queuing into the SSFNet sockets. All this enabled us to concentrate on Gnutella-specific operations while coding the protocol at the application layer, without too much interference from SSF-specific issues at the TCP layer.

Now, we will give a detailed description of how we have implemented the various parts of Gnutella protocol in SSF. We begin with the core protocol services of Gnutella, followed by its more advanced features. The simulation framework has been programmed to log all relevant actions in a log file. At each stage of our description, we give an example how the action appears in the log file.

4.1 Core Protocol Services

This section explains the architecture of standard Gnutella messages, along with the basic proto-

col services like initialization, bootstrapping, handshaking, querying, file search and file transfer.

While we have chosen to use Gnutella as our example P2P system, we stress that we want to find results applicable to P2P systems in general. Hence, features which are too specific to Gnutella protocol, and do not impact our investigation aims directly, have not been accorded too much attention.

4.1.1 Initialization

Each servent is assigned an IP address by the SSF based on the network topology (specified by the DML file). On starting the simulation framework, each servent outputs a few status lines about itself, briefly describing the number of files and their total size that it shares, and whether it has FlowControl, PongCaching and Ultrapeer capability enabled. The initialization phase appears in the log file as follows.

```
SimTime[0.0] Source[0.0.20.74] Msg:97 Files and
774298 KByte
SimTime[0.0] Source[0.0.20.74] Msg:FlowControl:No
SimTime[0.0] Source[0.0.20.74] Msg:PongCaching:Yes
SimTime[0.0] Source[0.0.20.74] Msg:Ultrapeer:No
```

After this, a servent opens its listening socket. This implies that the servent is ready to participate in the Gnutella network.

```
SimTime[0.0] Source[0.0.20.74] Msg:Listening
Socket: START
```

4.1.2 Bootstrapping

In order to connect to the Gnutella network, a servent needs as a starting point, the addresses of some hosts within the Gnutella network.

There are various possibilities by which a servent may obtain a host's address.

1. By accessing a *GWebCache*: an HTTP request is sent to a known GWebCache server, which responds with a set of nodes.

2. The *X-Try* and *X-Try-Ultrapeers* headers used during handshake process contain some hosts' addresses. These addresses tend to be of a good quality.

3. Retrieving addresses from *Pong* or *QueryHit* messages. However, this requires that the servent has atleast one and two connections already established within the network respectively.

Each servent maintains a private cache, called the *HostCache*, where it stores the addresses of other hosts in the network, which it learns over time in a variety of ways. The recommended method of bootstrapping for each servent is to use its own *HostCache* in conjunction with the *GWebCache*.

We implement a slightly simplified method for Bootstrapping. All servents, on starting up, register themselves (their IP address and port) at a globally unique central *Manager*. The *Manager* stores the servent addresses in its database. Our *Manager* is a rough abstraction of the *GWebCache*.

When a servent wants to connect to the Gnutella network, it makes a function call to the *Manager*, which returns a set of upto 10 randomly chosen servent addresses from its database. The servent initializes its *HostCache* with this set of addresses returned by the *Manager*, and uses it to connect to the Gnutella network.

4.1.3 Handshaking

The servent selects a random address from its *HostCache*, and attempts to establish a TCP connection to this servent.

```
SimTime[7.37] Source[0.0.20.74:5001] Target[0.0.16
.6:6346] Msg: sending TCP Connect
```

On successful TCP connection establishment, the two servents exchange Gnutella headers, as outlined below.

```
SimTime[7.55] Source[0.0.20.74:5001] Target[0.0.16
.6:6346] Msg: sending GNUTELLA CONNECT/0.6
SimTime[7.65] Source[0.0.16.6:6346] Target[0.0.20
.74:5001] Msg: sending GNUTELLA/0.6 200 OK
SimTime[7.74] Source[0.0.20.74:5001] Target[0.0.16
.6:6346] Msg: sending GNUTELLA/0.6 200 OK
```

At this stage, the two servents have successfully established a Gnutella connection between them.

4.1.4 Standard Message Architecture

The servent, after connecting to the Gnutella network, corresponds with other servents using Gnutella messages. The different types of Gnutella messages have already been introduced in Section 2. Now, we detail the architecture of each message as implemented by us.

While in reality, one IP packet may contain several Gnutella messages, and one Gnutella message may be split up among multiple IP packets, we have simplified the implementation by assuming that each IP packet contains only one Gnutella message. Besides, we use only IPv4 addresses in our simulation framework.

4.1.4.1 Message Header

The message header, which is common to all Gnutella messages, is 23 bytes long, and is composed of the following fields.

- Message ID/GUID (Globally Unique ID)

- Payload Type
- TTL (Time To Live)
- Hops
- Payload Length

Message ID: It is a 16-byte string, called GUID, which uniquely identifies any message in the network. It contains all 1's in byte 8, and all 0's in byte 15. The values for other bytes are generated randomly.

Payload Type: It indicates the message type. It can have the following values: Ping, Pong, Bye, Route Table Update, Push, Query, QueryHit.

One may notice, that we have implemented one additional message that is not specified by the RFC. The message, Route Table Update, is used in the Query Routing Protocol, for communicating the routing table updates from the leaf node to the ultrapeer.

TTL: It indicates the number of times a message will be forwarded by Gnutella servers, before being discarded from the network. Each server decrements the TTL before forwarding the message. When the TTL value for a message reaches 0, it is not forwarded any more.

Hops: It indicates the number of times a message has already been forwarded. At any time during its journey within the Gnutella network, a message must always satisfy the following condition:

$$TTL(0) = TTL(i) + Hops(i)$$

where $TTL(i)$ and $Hops(i)$ are values of TTL and Hops field of a message at any particular instant, and $TTL(0)$ is the maximum number of hops a message can travel (set to 7).

Payload Length: It denotes the length in bytes of the actual message following the header. Since there are no pads between Gnutella messages, this field also denotes the beginning of the next message.

The maximum size of a Gnutella message is 4 kB. The TTL and Payload Length fields, being critical to proper Gnutella operation, are checked rigorously. The message header is followed by the message payload, which can be of the following types.

4.1.4.2 Ping This message contains no payload. It looks as follows.

```
SimTime[7.83] Source[0.0.20.74:5001] Target[0.0.16
.6:6346] Msg:sending GnutellaMsgPing[]:
START; MessageID: 65a9b2
```

4.1.4.3 Pong This message gives information about the number of files, and their total size, that a server is sharing on the network. Its message ID is the same as that of the Ping to which it is a response. It looks as follows.

```
SimTime[7.92] Source[0.0.16.6:6346] Target[0.0.20
.74:5001] Msg:sending GnutellaMsgPong[Shared
Files: 97; Shared KBytes: 774298]: START;
MessageID: 65a9b2
```

4.1.4.4 Query The total size of this message should not exceed 256 bytes. It contains two fields: the minimum speed (in kB/second) and search criteria. A server receiving a Query with minimum speed of n kB/second should only respond with a QueryHit if it can communicate at a speed $\geq n$ kB/s. The search criteria indicates that the Query is searching for file names that contain the particular search string. A Query looks as follows.

```
SimTime[61.18] Source[0.0.20.74:5001] Target[0.0.16
.6:6346] Msg:sending GnutellaMsgQuery[MinSpeed: 10
;searchString: Govind]: START; MessageID: d18e74
```

4.1.4.5 QueryHit This message contains the following information: the number of files the host possesses that match the corresponding Query's search criteria, and the speed of the host in kB/second. The message also contains a Result Set, which is a set containing some information about each of the matching files like its index, size, and name.

The IP address and port of the server are also part of the message. The port indicates the port number on which the server can accept incoming HTTP file requests (for actual file downloads). It is the same as that used for Gnutella network traffic. Its message ID is the same as that of the corresponding Query message. A QueryHit message looks as follows.

```
SimTime[61.27] Source[0.0.16.6:6346] Target[0.0.20
.74:5001] Msg:sending GnutellaMsgQueryHit[Number:
9; Speed: 10]: START; MessageID: d18e74
```

4.1.4.6 GGEP Gnutella Generic Extension Protocol (GGEP) allows arbitrary extensions to Gnutella messages. A GGEP block is used to specify additional information or instructions in any Gnutella message. For example, it can be used to provide details on how to parse the search criteria in a Query, or to provide additional information about a file in the Result Set of a QueryHit. We have provided GGEP support for Gnutella messages in our simulation framework. This gives us the ability to extend our code at a later time, when we need to experiment with special features of Gnutella.

4.1.5 Querying the Network

Here we give some general additional information regarding processing and routing of Ping/Pong and Query/QueryHit messages.

A servent forwards an incoming Ping/Query message to all of its directly connected servents, except the one from whom it received the Ping/Query message. There are some variations to this rule in case of servents using Flow Control, Pong Caching or Ultrapeers capabilities, which will be explained in Section 4.2.

A servent decrements the TTL and increments the Hops field of the message header before forwarding it. If after decrementing, the TTL equals 0, the message is not forwarded.

If a servent receives a Ping/Query with the same message ID as it has received previously, it discards the message, as it is a duplicate.

A Pong/QueryHit message is sent along the reverse path as that of the corresponding Ping/Query message respectively. Every servent implements a forwarding table, where for every Ping/Query message forwarded, a table entry is stored. The table entry uses the message ID as the key, and the servent connection from which the message arrived as the value. When the servent receives a Pong/QueryHit, it looks up its message ID in the forwarding table. If the servent has seen the corresponding Ping/Query message, it would find the message ID in the forwarding table (a Ping/Query and its corresponding Pong/QueryHit have the same message ID respectively). In this case, the servent will forward the Pong/QueryHit to the servent connection stored in the forwarding table. Otherwise, the Pong/QueryHit is not supposed to traverse this path, and is hence removed from the network.

4.1.6 File Search

The basic scheme is that, when a servent receives a Query, it checks its own file contents to determine if it has any matching entries for the search criteria of the Query. If there is a match, the servent composes the result in the Result Set of a QueryHit message, and sends it along the reverse path of the corresponding Query.

We keep a centralized list of all the file names used in the simulation framework in an ASCII file called *shared_resources.txt*. During the initialization phase, all the servents participating in the network are assigned a set of files from this centralized list. To improve the run-time performance of file search operation, we use a HashSet [10]. A HashSet is a Java class that implements a kind of a set, backed by a hash table. It offers constant time performance for basic operations like adding/removing

elements and testing for existence. For each servent, we compute a HashSet of the file names possessed by it during the initialization phase. When a new Query is generated during simulation, the *Manager* (see Section 4.1.2) computes a HashSet of file names contained in *shared_resources.txt* that match the Query. When the Query arrives at any servent, the HashSet of the Query is intersected with the HashSet of the servent. If the servent possesses any files satisfying the Query, this information is passed into the Result Set of the QueryHit message.

The result HashSet of each Query is cached at the *Manager*. Hence, when a new Query is generated with the same search criteria, the resulting HashSet can be reused, thus making the processing chain of the new Query much faster.

4.1.7 File Transfer

When a querying servent receives a QueryHit response, it can initiate the download process for any of the files specified in the Result Set field of the QueryHit message. The actual file transfer takes place outside the Gnutella network, i.e. the source and target servents establish a direction connection between them using HTTP.

The servent desiring a file (source) initiates a download request to the servent possessing the file (target) using the GET command of HTTP. The requested file's index and name, gleaned from the Result Set of QueryHit message, are passed as parameters. The download request looks as follows.

```
SimTime[67.07] Source[0.0.20.74:5001] Target[0.0.16
.6:6346] Msg:sending HTTP-GET[Index: 1742; Name:
Shri Krishn Govind.mp3]: START;
```

The target servent responds to this with HTTP compliant headers, followed by the actual file data. It appears in the log file as follows.

```
SimTime[67.39] Source[0.0.20.74:5001] Target[0.0.16
.6:6346] Msg:sending HTTP-Data[Length: 3453123,
Data of file index: 1742]: START;
```

This completes the file transfer from the target to the source servent.

4.2 Advanced Features of Gnutella

In this section, we discuss advanced properties of Gnutella protocol like ultrapeers, Query Routing Protocol, flow control and Pong Caching.

4.2.1 Leaf mode and Ultrapeer mode

In the initial version of Gnutella (version 0.4) [4], all nodes connected to each other randomly. But to enable Gnutella to scale, a hierarchical structure was developed, whereby nodes were classified

into leaves and ultrapeers. A leaf can only make a small number of connections, and that only with ultrapeers. An ultrapeer, which is a servent with high processing and connection capacity, connects to a large number of leaves, and a few other ultrapeers. It acts as a proxy for its leaves to the Gnutella network. An ultrapeer forwards a query to a leaf only if it believes that the particular leaf can answer that query. A leaf, on its part, only communicates with its ultrapeers, and with no one else. This has the overall effect of reducing the number of nodes involved in message handling and routing, and more importantly, of reducing the traffic in the network. Besides, it allows servents with lesser processing/connection capacities (leaves) to participate in the system in an effective manner, without being overwhelmed by machines with superior capabilities.

Each servent is assigned to be a leaf or an ultrapeer during the startup phase of the simulation, through the DML file definition corresponding to each servent. This allows us better control over the simulation setup and topology.

An ultrapeer uses the Query Routing Protocol (QRP) to decide which query to forward to which of its leaf nodes. Now we explain how we implemented the QRP.

4.2.2 Query Routing Protocol

QRP governs how an ultrapeer filters incoming **Queries**, and forwards them to only those leaf nodes, which are likely to match the **Queries**. The leaf nodes send a Query Routing Table (QRT) to the ultrapeer, and the ultrapeer makes its decisions by looking up these routing tables.

It is important to note that the aim of QRP is to avoid forwarding **Queries** that cannot match, rather than to forward only those **Queries** that will match.

The protocol operates at two levels: at the leaf node, and at the ultrapeer.

At the leaf node:

1. We break the resource names into individual words. A word is a consecutive sequence of letters and digits.

2. We hash each word with a hash function as described in [5] and insert a “present” flag in the corresponding hash table slot. The hash table is a big array of bits, and we do not store the key, but only the fact that the key ended up filling some slot. Before hashing, we convert all words to lower-case, and remove all accents. Besides, we remove all words less than 3 characters in length.

3. We then remove the trailing 1, 2 and 3 characters of each word, and re-hash the 3 new words formed in this way, provided their length

is greater than 2 characters. This is done with the aim of removing plurals and word-endings like ‘ing’, ‘ed’, etc. from words.

As an example, consider the file name “Bhajo Gopala.avi”. This will give rise to the following hash table entries: bhajo, bhaj, bha, gopala, gopal, gopa, gop, avi.

4. When all the resources of a leaf are hashed, the complete hash table forms the QRT of the leaf. The QRT is optionally compressed, broken into smaller messages, and sent with the normal Gnutella traffic to the ultrapeer, in the form of Route Table Update messages.

The hash table we currently use is 1024 bits in size. This size was found to be sufficient for our current experiments, but can be easily increased on demand.

All the leaf nodes thus build their routing tables and send it to their ultrapeers. If the file contents of a leaf node change, the routing table updates are sent to the ultrapeer in the form of Route Table Update messages.

At the ultrapeer:

The ultrapeer stores the QRTs of each of its leaf nodes. On receiving any **Query**, the ultrapeer breaks the search string into individual words, and makes a hash table lookup for those individual words in the QRT of each of its leaf nodes. On finding a match, the ultrapeer forwards the **Query** to that particular leaf node.

4.2.3 Flow Control

It is a mechanism used to regulate the amount of data that passes through a connection. The overall scheme has been implemented as follows. There are 4 input queues for each servent connection, corresponding to each Gnutella message type. All incoming messages are queued in their respective queues. Each servent has been assigned a pre-decided output bandwidth of 10 kB/second for sending messages. The message queues are processed in FIFO (first-in-first-out) order. Individual messages are prioritized (from most to least priority) as: **QueryHit**, **Pong**, **Query**, **Ping**.

In other words, the **QueryHit** queue is processed first, in FIFO order. All its messages are forwarded one-by-one. Next, the **Pong** queue is taken up, and so on, until all the queues are empty or the output bandwidth of 10 kB/s is fully used up.

To limit excessive data, if the total amount of data in all input queues exceeds 10 kB, all **Queries** which are not originating at the servent itself are dropped. This is done to avoid queuing back potentially large results for these **Queries** when we are

already facing a throughput problem.

The HTTP file transfer (which actually takes place outside the Gnutella protocol) is also flow-controlled. It is guaranteed a minimum data flow rate of 1 kB/s, while the maximum allowed rate is 10 kB/s. Hence, irrespective of the number of Gnutella messages in the input queues of a server, the HTTP data will be accorded a flow rate of at least 1 kB/s. However, if the output bandwidth of 10 kB/s is not being fully used by a server for sending Gnutella messages, the HTTP data will be allowed to use up the remaining bandwidth. All this is done because the entire purpose of Gnutella is to allow peers to exchange files. The various Gnutella messages have been designed to facilitate this one purpose. Hence, the actual file transfer operation should not be penalized at the cost of Gnutella message transfer.

4.2.4 Pong Caching

Due to the broadcast effect of Pings, the amount of traffic generated by Ping/Pong messages in Gnutella is immense [6, 9]. To limit the number of Ping/Pong messages in Gnutella, a scheme called Pong Caching has been introduced. A number of Pong Caching proposals can be found at [7].

Our implementation for Pong Caching works like this. A server sends a Ping to all its neighbors every 4 seconds, with a TTL of 7. For each connection, a server maintains a cache, where it stores a maximum of 10 most recent Pongs that it has received from that server. When a server receives a Ping, it replies it with upto 10 Pongs. These Pongs comprise of its own Pong, and upto 9 other Pongs that it randomly chooses from the caches that it maintains for each of its directly connected servers. In this selection process, it naturally does not include the server from which it received the Ping. It is noteworthy that the server does not forward the Ping to any of its neighbors.

There are two special Pings that have to be dealt with differently. Pings with TTL of 1 and Hop Count of 0 or 1 are replied by the server's own Pong only.

Pings with a TTL of 2 and Hop Count of 0 are called Crawler Pings. A server uses them to probe the network for its neighbor's neighbors. When a server receives such a Ping, it is supposed to answer with the Pongs of its directly connected neighbors. We have implemented a separate cache for each server, where it stores the Pongs of its immediate neighbors. These Pongs are gathered by sending out Pings with TTL of 1 and Hop Count of 0 or 1.

One can notice that in our implementation, a server never requires to forward any incoming Pings,

irrespective of its TTL. It can answer all Pings from its caches, which are maintained with fairly healthy (recently active) Pongs at all times. This is possible due to each server sending a Ping to its neighbors at regular intervals.

5 Correlation of P2P structure to simulated topology

In this section, we explain how we correlate a P2P structure to the simulated topology in our simulation framework. The network topology to be simulated is specified with the help of a DML file. SSFNet automatically assigns IP addresses to all host and router interfaces specified in our DML network model. The IP addresses are aggregated in blocks according to the CIDR (Classless Inter-domain Routing) recommendations. A detailed explanation of automatic assignment of IP addresses to DML network models by SSFNet can be found at [8].

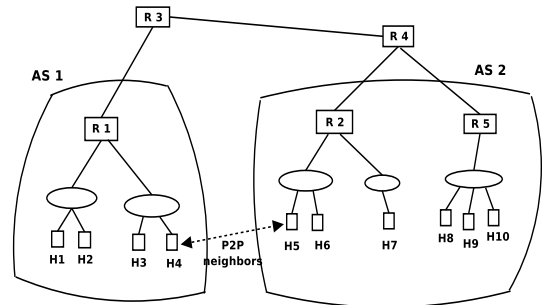


Figure 1: Topology correlation

Consider Figure 1 as an example topology. The network shows two autonomous systems, AS1 and AS2. The bigger rectangles denote routers, the ovals denote LANs, and the smaller rectangles denote individual hosts. Each of the hosts and router interfaces are assigned IP addresses by SSFNet. Lets assume that all the hosts are participating in the P2P network. Suppose hosts H4 and H5 establish a P2P connection between them. This implies that H4 and H5 are P2P neighbors, at an application layer distance of 1. However, the actual network path between them goes through R1, R3, R4, and R2. Considering the number of router hops as a metric for network layer distance, H4 and H5 are at a network layer distance of 5. Considering AS hops as a metric, the two hosts are at a network layer distance of 2. Hence, we see that two hosts, while being neighbors at the P2P (application) layer, have a different path (and distance) at the network layer.

6 Validation of simulation framework

Now, we would like to discuss some work-in-progress: this phase of our research focuses on monitoring the performance of our simulation framework, and amending it continually to reflect real-world behaviour as closely as possible.

Having verified that the system adheres to the protocol specifications, we calculated the proportion of different Gnutella messages in the total Gnutella traffic, and found that our statistics were similar to real-world behaviour. We investigated the amount of traffic reduction achieved by enabling Pong Caching and Ultrapeer capabilities. We found out that there is indeed a significant reduction in total Gnutella traffic on enabling these features. We also found that the amount of traffic reduction is heavily dependent on the parameters of the Pong Caching scheme used, and the topology of the simulated network. When we experimented with simple networks comprising only a few ASes (say 3 or 4), the traffic reduction was not huge. But as the number of ASes and the size of network grows, the traffic reduction becomes more significant. Consolidating this area is a major focus of our future research. We will experiment with different realistic topologies, along with various parameters for Pong Caching, etc to come up with accurate statistics.

To enable close monitoring and extensive experimentation, we have made our simulation framework heavily configurable. We can easily configure various different parameters at different levels of our framework. For example, we can change the number of Pongs returned by a server, the input message queue size of a server, the rate of sending Pings, etc. We can bring more variety in the files shared by the servers, and the rate and content of Queries generated. Facilitated by our comprehensive logging capabilities, we are in the process of calibrating the effect of all these parameters on the Gnutella network.

Another direction of experimenting is measuring the impact and comparing the effectiveness of different algorithms for Pong Caching, flow control and Query caching. We also intend to augment our simulation framework with a dynamic user behaviour model. In real world P2P systems, nodes join and leave the network in a highly dynamic manner, with varying session durations. We aim to reflect this behaviour in our framework. This task will involve investigation of real Gnutella logs, and interpreting the data in a manner that can be implemented by us in our simulation framework. This could later be extended with a dynamic file distribution model of P2P users.

7 Summary

We have provided a detailed explanation of the mechanism used to implement a multi-layered, highly structured, heavily configurable simulation framework for the Gnutella P2P system. The in-built support for GGEF in Gnutella messages allows arbitrary extension of the protocol, and provides the ability to experiment extensively with different features of the P2P system. The general implementation has been so designed that it leads to a faster and more memory efficient execution, e.g. during file search and QRP processing. Another good feature of the framework is the guaranteed minimum bandwidth for actual file transfers achieved by flow control. In short, we have presented a simulation framework that gives us the ability to experiment with P2P systems in an efficient manner.

References

- [1] RFC. *Gnutella 0.6* <http://rfc-gnutella.sourceforge.net/developer/testing/index.html>
- [2] Slyck. <http://www.slyck.com>
- [3] Scalable Simulation Framework. <http://www.ssfnet.org/homePage.html>
- [4] RFC. *Gnutella 0.4* <http://rfc-gnutella.sourceforge.net/developer/stable/index.html>
- [5] LimeWire. *Query Routing for the Gnutella Network*. http://www.limewire.com/developer/query_routing/keyword%20routing.htm
- [6] M. Ripeanu, I. Foster and A. Iamnitchi. *Mapping the Gnutella Network*. IEEE Internet Computing Journal, 2002
- [7] Pong Caching schemes. <http://rfc-gnutella.sourceforge.net/developer/testing/pongCaching.html>
- [8] SSF. *IP Addresses in SSFNet*. <http://www.ssfnet.org/InternetDocs/ssfnetTutorial-1-vlsm.html>
- [9] Christopher Rohrs. *LimeWire's Pong Caching Scheme*. <http://rfc-gnutella.sourceforge.net/src/pong-caching.html>
- [10] Java Docs. <http://java.sun.com/j2se/1.4.2/docs/api/java/util/HashSet.html>
- [11] S. Saroiu, K. Gummadi and S. Gribble. *A Measurement Study of P2P File Sharing Systems*. Multimedia Computing and Networking, 2002
- [12] V. Aggarwal, S. Bender, A. Feldmann and A. Wichmann. *Methodology for Estimating Network Distances of Gnutella Neighbors*. GI Informatik - Workshop on P2P systems, 2004