Published in Lecture notes in computer science 2468, 255-257, 2002
which should be used for any reference to this work

Experimenting with Gnutella Communities

Jean Vaucher[1], Gilbert Babin[2], Peter Kropf[1], and Thierry Jouve[1]

[1] University of Montreal, Computer Science and Operations Research, Montreal, Quebec, Canada
[2] HEC – Montreal, Information Technology, Montreal, Quebec, Canada
{vaucher,kropf,jouvethi}@iro.umontreal.ca, Gilbert.Babin@hec.ca

Abstract. Computer networks or distributed systems in general may be regarded as communities where the individual components, be they entire systems, application software or users, interact in a shared environment. Such communities dynamically evolve with components or nodes joining and leaving the system. Their own individual activities affect the community's behavior and vice versa. This paper discusses various practical experiments undertaken to investigate the behavior of a real system, the Gnutella network, which represents such a community. Gnutella is a distributed Peer-to-Peer data-sharing system without any central control. It turns out that most interactions between nodes do not last long and much of their activity is devoted to finding appropriate partners in the network. The experimental results presented have been obtained from a Java implementation of Gnutella running in the open Internet environment, and thus in unknown and quickly changing network structures heavily depending on chance.

1 Introduction

Whenever autonomous individuals act in a shared environment, the emergent interaction results in manifold relations between the individuals and/or groups of individuals. Those relations and the associated behavior of individuals and groups may induce structures to the groups. Such structures are commonly called *communities*. The behavior of biological individuals, such as ants or bees, but also humans has been widely studied in the social sciences [4]. Key findings include that despite the largely varying (intellectual) capacities of individuals and groups, a set of common characteristics for acting in a shared environment still may be observed [9]. However, this usually depends on the specific knowledge of the individuals and their time already spent within a community. Among the characteristics identified, we find that,

- each individual can identify a few members of a community and may exchange information with them;
- there is no single individual who knows or controls the whole community;
- some individuals may be more "intelligent" than others and have more and/or better information;
- communities are often hierarchically structured with one or more outstanding individuals.

Starting with just a few individuals (at least two), communities continuously evolve. The resulting set of inter-related community members is generally called the *social network* of a community. While the number of individuals in a community can grow very fast, the single individual needs only little information about other individuals to still be able to potentially interact with a large number (or all) of the community members. The *six degrees of separation* property [14] illustrates this in the case of human communities. Moreover, communities are often characterized by a highly self-organizing behavior. Insect collectives such as ants or bees, but also physical and chemical systems composed of large numbers of individuals or particles interact locally and contribute thereby to global organization, optimization and adaptation to the environment.

Computer networks or distributed systems in general may be regarded as communities similar to the above examples. Most obviously, the Internet or Web forms entities that can be characterized as communities. Many approaches to define communities on the Web [6, 8, 10] are based on the use of existing link patterns and they therefore lack the characteristic community properties to adapt to the current context and to dynamically evolve. Implicit information [9, 13] other than link patterns are necessary to achieve this.

A number of applications have been developed which include in one way or another the idea of communities on the Internet. Among those are Yenta [7], an agent based system to find people with similar interests and to make them known to each other, Freenet [5], an information publication system storing, caching and distributing information on demand without any centralized control, or Gnutella [2], a distributed Peer-to-Peer data-sharing system. In the Gnutella system, a user only needs to know one (or several) other participants to join the community. The mechanism to broadcast information (search for partners or particular data) refrains from any central control: it is based on propagation of messages from participant to participant. The community is highly dynamic as participants can join and leave at any time without having to contact any administrative unit.

In order to investigate the behavior of communities on the Internet, the Gnutella system has been chosen for the research presented in this paper. The system provides an ideal practical testbed because all the participating individuals are unknown, no central control exists, and the community is sufficiently large. In fact, the only common component is the communication protocol and the core system behavior where each participant acts as a client and a server at the same time while applying the aforementioned information propagation mechanism.

The next two sections introduce the Gnutella system and protocol. In section 4 we describe the Jtella platform used for the experiments which are presented in detail in section 5. The method applied for measuring the performance of Gnutella applications and the results observed are discussed in section 6 before concluding with a discussion about the various experimental results observed.

2 Overview of Gnutella

Gnutella is a distributed Peer-to-Peer (P2P) application for the sharing of files over the Internet. It was designed as a replacement for Napster [21] and has been used mainly for the dissemination of multimedia files.

Each participant in a Gnutella network runs a program on his computer that acts both as a client and a server as well as a router. Gnutella programs are referred to as *servents* (SERVer+cliENT), *nodes* or simply *clients*. As a client, the application provides an interface where a user can enter keywords describing the files that he is seeking. The program then sends the request to neighbouring participants who pass it on to their neighbours who do the same; thus propagating it throughout the network. At the same time, clients check to see if the request corresponds to local files they are willing to share and, if so, they send back a response. File transfers are done via another route using standard HTTP protocol requests.

The fundamental feature of Gnutella is that it does not rely on centralized databases or proprietary software. It also tries to ensure a measure of anonymity. As a result, it is resistant to both hardware failure and legal attack. The first Gnutella application was released in March 2000 but it was officially available for only a 24 hour period [2]. The basic protocol implicit in the original software is quite simple and is now available on the Web [1]. Although, it has been reported to suffer from performance and scalability problems [18], the Gnutella protocol has resulted in a large number of implementations.

Initially, as a replacement for Napster, the Gnutella network grew exponentially and this growth has been charted by several researchers [3, 17]. Available data shows the network growing from around 1,000 nodes in November 2000 to over 40,000 in June 2001. Over this period, Ripeanu [17] found that over 400,000 different users had connected to Gnutella. In another study, a *crawler* programme found over 1 million different host addresses in an 8 day period [20]. However, since the summer of 2001, the network has been steadily shrinking, reaching an average of 16,000 users in January 2002 [3]. One can surmise that, if the main interest in Gnutella was sharing of music, many users have switched to more efficient specialised services such as Morpheus-KaZaA from MusicCity which now claims to have over 300,000 simultaneous users [15].

As the first widespread decentralised and open protocol, Gnutella is worthy of study. Its simple basic protocol also make it easy to use in experiments. Because the protocol is not specifically oriented to a single application domain (like mp3-encoded music), it is also easy to use Gnutella as a low level dissemination or broadcast protocol upon which to piggy-back other applications — with specially formatted query/response strings. Parallel private Gnutella networks can also be set-up by the simple expedient of using private bootstrap host caches.

3 Description of the Gnutella Protocol

Each participant in a Gnutella network maintains a small number of permanent links to neighbours (typically 4 or 5). Search is done via flooding — a distributed form of broadcast. Messages are sent to the neighbours who pass them on to their neighbours

and so on. The number of hosts which are contacted in this way increases exponentially with each jump. In order to limit the potential data explosion, the number of jumps is bound by a time-to-live (TTL) counter which is decremented on each passing on. When the counter reaches zero, the message is no longer propagated. Messages also have a hop counter to keep track of how far they have come.

Gnutella provides also for some notion of anonymity. Specifically, queries do not contain the identity of the initiating host. Instead, each Gnutella message has a unique identifier (ID) and propagating hosts maintain routing tables keyed on this ID which indicate from which connection a message arrived. Answers carry the same ID and are returned along the same route as the query. The anonymity is only relative because downloads are done directly without passing through the Gnutella connections. The routing tables are also useful in preventing looping and duplicating messages: if the ID of a query (not an answer) is already present in the table, the message is seen to be a duplicate and is not propagated.

The Gnutella protocol is based on four types of messages (actually there is a 5^{th} type to deal with firewalls, but it is not pertinent to our discussion). The messages come in pairs: one for the requests and one for the answers. The file search pair includes:

Query – contains the user request as an unformatted string of keywords [3];
Reply – used by a host to return a list of matching files along with a short description of each file as well as the Host:Port address to be used for an HTTP download.

The next two messages are used to discover the addresses of participating hosts:

Ping – a request for host addresses;
Pong – a reply to the Pong with a Host:Port address along with extra information about the host bandwidth and the number of local files.

According to the protocol, a host receiving a Ping should answer with its own address in a Pong as well as forwarding the Ping to its neighbours. In practice, to reduce overhead, a host which already has too many neighbours, may pass on the message without returning its own address. Some hosts may act as central directories. They do not propagate Pings; rather, they maintain a cache of addresses they have received and return a small number of these. A number of sites well-known to the Gnutella community act as directories and this is how initial connection to Gnutella operates. However, any active host can serve as an initial connection point.

Finally, the protocol gives details about the handshake to be used on initial connection and suggests that Gnutella applications use port 6346 as the server address. The other aspects of the functionality that we expect in any program that accesses the Gnutella network either rely on other protocols or are left unspecified.

4 The Experimental Platform

Our experimental platform is based on Jtella, a framework written by Ken McCrary [11, 12]. Jtella is made up of about 40 classes and 7000 lines of Java. It manages the initial

[3] Note: in possible extensions of Gnutella to specialized areas, one would expect the format and semantics of the Query request to be more tightly defined.

connection to the Gnutella network, the maintenance of a specified number of connections and the routing of messages. An indication of the ease of use is that simple applications to search or monitor traffic require less than 150 lines of Java (on top of the Jtella Framework). Note that Jtella does not cover the indexing of files, matching queries or media playing. Jtella served as our introduction to the implementation of the Gnutella protocol but we took the liberty of rewriting or modifying about 1000 lines mainly dealing with parallelism and synchronization. We also uncovered, reported [22] and bypassed a Java bug: threads which are not started are not garbage collected.

4.1 Architecture of Jtella

The main building blocks of Jtella are: the Connection objects, the Router, the Connection managers and the Host Cache. The architecture is shown in Fig. 1. It is quite similar to that of LimeWire [19].

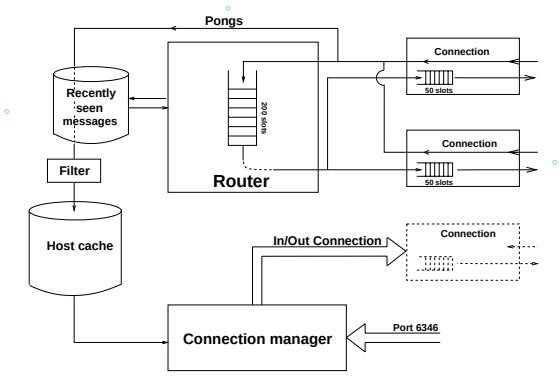

Fig. 1. Servent Architecture

There is one Connection object for each connection. Each Connection acts as a Thread to handle incoming messages and put them on a common message queue for the Router. Each Connection also has a second Thread with a message queue to handle output messages.

The Router is a separate Thread. It takes messages off its queue, checks them against a table of recently seen messages and, if they are not duplicates, it places them in the appropriate output queues. The router tables are also used to return answers. Queues are fixed in length: the router queue has 200 slots and the output queues are 50 elements long. In general their occupation is less than 5%.

There are two connection managers whose job it is to maintain a specified number of active connections. Initially, Jtella was set up to keep 4 connections open: 2 outgoing and 2 incoming. If an incoming connection failed, the following incoming connection would be accepted. If the number of outgoing connections was below the specified level, for every missing connection, the outgoing manager would launch 2 start-up threads to

try to connect to new hosts. The current version is more flexible in the split between incoming and outgoing connections. By default, two slots are reserved: one incoming and one outgoing, but the others can be of either type. When a connection fails, we launch two connector threads and, at the same time, we accept incoming connections. If all connection attempts succeed, we can temporarily have too many connections; but connections die quickly and this excess capacity is short lived. On start-up, all connections are necessarily outgoing but in a short time, as our address is made known through Pongs, the rate of incoming requests increases to the point that most failed connections are replaced by incoming connections.

To discover addresses of Gnutella participants, we send out a Ping whenever we open a new connection, and we put the addresses from all Pongs received into a cache. Because we receive many more host addresses than we can use, we limit our cache to 200 addresses and discard the others. As will be discussed later, most addresses that we receive are invalid. An important modification to Jtella was the addition of a filter to weed out bad addresses. When the cache is empty or low, we connect to well known host caches. These use the same Ping/Pong messages as all other Gnutella nodes but their sole function is to store addresses and redistribute them to later callers. Recently, this host cache function has been partially delegated to the network and in many servent implementation, whenever a node refuses a connection request, it sends back a number of Pongs from its host cache before shutting down the connection. Commonly, some servents send back 10 Pongs and others 50.

5 Gnutella Measurements

An important characteristic of Gnutella is that performance for any one session is highly dependent on chance. If a client happens to find reliable hosts early, it will obtain a steady flow of messages. At other times, it may struggle to find even a single permanent connection and it is not rare for identical servants run in parallel to have 2:1 differences in performance indicators.

Before proceeding to more exact measurements and tests, we present typical output from two exploration experiments which show the difficulty in maintaining connectivity and quantifying behavior.

5.1 Exploration Experiment I

Our principal measurement program named TestServent sets up a node with a specified number of connections, routes messages and collects statistics. It also prints out the current status of the node every 15 seconds. Typical output is shown in Figure 2.

The first lines show the average traffic since the previous printout. The first thing to notice is that while 84 messages per second were received, most were invalid (either duplicates or Pongs with incorrect addresses); this left 22 valid messages which gave rise to 38 output messages.

Next, we see a snapshot of the connection activity. These are listed in the order in which they were created. OUT connections are created by our client, whereas IN connections were initiated by other hosts. For each connection, we give a status code (i.e.,

```
*****************  Fri Jan 04 10:01:33 EST 2002 **********************

Traffic in:  84 msg/s.
Valid   in:  22 msg/s.
Traffic out: 38 msg/s.

             Msgs:  Ping/Pong  -> Que/Rep
OUT ( OK )   1891:  335/998 -> 540/18  - cc652-a.plnfld1.com:6346
IN* ( OK )   220:  71/76 -> 73/0 - ACB51A11.ipt.aol.com:6349
IN  (temp)   1:  1/0 -> 0/0      - dmitry-pc4.la.asu.edu:47260
OUT (->? )   0:  0/0 -> 0/0      - 172.16.10.30:6355
IN  (temp)   0:  0/0 -> 0/0      - d15103.upc-d.chello.nl:2298
OUT (->? )   0:  0/0 -> 0/0      - 24.45.210.203:6346
OUT (->? )   0:  0/0 -> 0/0      - 62.70.32.25:6346
OUT (->? )   0:  0/0 -> 0/0      - 172.133.132.111:6346

Host Cache: 200  ==> Received: 2354, valid: 1116, used: 239

Threads:     59
   - SocketFactory Threads:     38
```

Fig. 2. Partial Output of Program TestServent

OK), list the number of messages received (total then categorized), and finally give the address of the corresponding host. In this test, we were trying to maintain 4 active connections, but at this moment there are 8 connections with only two in normal operation (OK). The other 6 connections are in various states of initialization or termination.

The first line shows the oldest connection, which has received 1891 messages and has been in operation for about 25 seconds. Only 18 messages are replies to queries. Typically, more messages are concerned with maintaining connectivity (i.e., Pings and Pongs) than with searching for information.

The second line shows an active *input* connection. The "*" indicates that the connected host has responded to our Ping with a Pong reporting its public port number (6349). Of the other 6 connections, two, noted *temp*, are incoming connections which we decided to refuse (probably because 4 connections were up when they first arrived). We are keeping those temporarily open while returning some host addresses and waiting for them to answer our Ping. The other 4 are *output* connections in the process of opening a socket (`->?`).

The next line shows the state of the host cache. At present, it is full (200 addresses). 2354 Pongs were received but of those, less than half (1114) had valid distinct addresses. This number is more than enough because only 239 were used to open new outgoing connections or passed on to other nodes.

The last lines shows the parallelism (59 threads) required by Gnutella. It also underlines a problem with Java 1.3 whereby a thread trying to open a socket (our *SocketFactories*) may be blocked for up to 13 minutes before failing. In this case, the 38 Socketfactory threads include 34 blocked threads in addition to the 4 (`->?`) in the active list. The other 21 threads are involved in managing connections and routing messages.

This brief look at Gnutella underlines a fundamental aspect of the network: most connections do not last long and much of a client's activity is dedicated to finding replacements. In later sections, we will study this aspect more closely.

5.2 Exploration Experiment II

Figures 3 and 4 illustrate the stochastic nature of Gnutella. We ran two Gnutella sessions in parallel for 45 minutes and monitored two parameters: the number of messages received per second and the *horizon*, a measure of network size. More precisely, every minute, we broadcast a Ping and then we tally all the answering Pongs over the next 60 seconds.

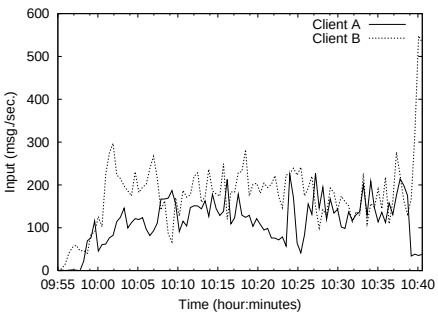

Fig. 3. Input Flow vs. Time for Two Clients

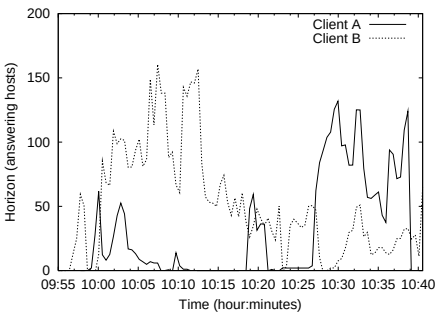

Fig. 4. Horizon vs. Time for Two Clients

Figure 3 shows the input rate while Figure 4 shows the horizon for the two clients. Both clients attempt to keep 4 connections open. The graphs are quite noisy but it is clear that Client B has done better than Client A. The average message rate for B is around 180 compared to 120 for A.

The random nature of operation is even more pronounced in the measurement of the *horizon*. It is hard to believe that these results were obtained from two *identical* programs run under *identical* conditions. This also shows the difficulty in trying to estimate the size of the Gnutella network.

In what follows, we present the results from experiments designed to quantify some critical aspects of Gnutella operation, namely:

- the validity of organizational information exchanged between nodes,
- the success rate in connecting to the network and
- the duration of sessions during which members actively participate in the network.

The experiments are presented in historical sequence as we tried to elucidate strange behaviour or make the implementation more efficient and robust. In all cases, we present results from 2 or more experiments to give an idea of typical behavior as well as variability. Although it is difficult to obtain exact meaningful measures of performance, our results still lead to interesting conclusions.

Later on, in Section 6, we present an experimental protocol that we used to overcome the stochastic nature of Gnutella in order to show the effect of an operating parameter, the number of active connections, on performance.

5.3 Validity of Pong Addresses

Open systems must deal with information received from many sources of untested quality. In the case of Gnutella, hosts depend on the Ping/Pong mechanism to discover the addresses of participating hosts. Unfortunately, early trials showed that many (if not most) addresses provided by Pongs are useless.

First, many addresses are duplicates. In experiments done around Oct. 16th, 2001, between 75 % and 88 % of addresses received were identical to addresses already present in our cache of 200 addresses. In one trial with 2390 Pongs received, a single address (32.101.202.89) was repeated 210 times. Fortunately, a simple test can be used to eliminate a large proportion of duplicates because about 25 % of addresses are identical to the one received immediately before.

Secondly, many addresses are "special" values (i.e. 0.0.0.0) which are obviously invalid. There are also blocks of Internet addresses which are reserved for particular uses and make no sense in the context of the Gnutella network. One example is multicast addresses but the most common problem results from hosts operating on private internets with NAT (Network Address Translation) translation [16]. These use addresses (i.e. 10.0.0.xx) which have no global validity. Table 1 shows the results from 2 experiments where we collected and analyzed all Pong addresses.

Table 1. Classification of IP Addresses Received in Pong Messages

	Experiment A		*Experiment B*	
Total addresses received	7482		19484	
Invalid addresses	2240	(30 %)	7042	(36 %)
Repeated addresses	1432	(19 %)	5298	(27 %)
Already in cache	1696	(23 %)	3978	(20 %)
Retained	2114	(28 %)	3166	(16 %)
Unique good addresses	1514	(20 %)	1792	(9 %)

As a result of these experiments, we modified the cache algorithm in our client to filter out invalid and repeated addresses as well as those already in the cache. With these mechanisms in place, the data above shows that between 16 % and 28 % of addresses are retained. Due to the limited size of the cache, not all duplicates can be detected. The last line of Table 1 — the results of off-line analysis of all addresses received — shows the actual proportion of unique valid addresses to vary between 9 % and 20 %. Even with this drastic filtering and the use of a small cache, we normally receive many more addresses than we need.

```
Host Cache: 197  ==> Received: 59719, valid: 17489, used: 4145
```

5.4 Creating Sockets

Having filtered out invalid addresses, we then considered the probability of success in connecting to hosts whose addresses we retained. There are several reasons why a

connection attempt could fail: the host may be too busy and refusing connections, the application may have terminated or the computer been disconnected from the network.

As mentioned previously, threads trying to open sockets to unavailable hosts remain blocked until the local system provides a timeout. In our set-up (Java 1.3 and Linux 2.2.17) this can take up to 790 seconds. In Windows environments, a smaller time-out of 45 seconds was reported. This delay has been a major source of inefficiency in both crawlers and servents; but the problem has been fixed in Java 1.4.

In one 90 minute session, our servent attempted to connect to 2541 hosts. Here is the breakdown of the results obtained and the average time to set-up the connection:

- 31 %: success - connection achieved in 2.3 sec.,
- 20 %: failure reported rapidly in 1.7 sec.,
- 49 %: blocked, failure noted after 10 sec.

To study this phenomenon more closely , we created a Connection tester (CTester) that takes a list of host:port addresses and tries to open a socket to each — which it then closes without attempting to do a Gnutella handshake. For each connection, it prints out the time until the socket creation terminates as well as the error message if the socket could not be created. For this test, we used 100 random addresses taken from a TestServent log file. Here are the results:

- 36 %: socket created in 1.6 sec. (9 sec. maximum),
- 26 %: rapid failure in 0.9 sec.,
- 38 %: blocking and failure reported after 790 sec.

The blocking during socket creation in Java explains the difficulty reported by several researchers who implemented crawlers to analyze the topology of Gnutella. Given the data above, where roughly, one connection attempt in three is blocked for 13 minutes, this means that a single thread can only examine about 4 addresses/minute and multi-threading is obviously a must.

5.5 Duration of Connections

We analyzed the log files from several sessions to determine how long connections stay valid once they have been established. In our longest test, maintaining around 7 connections over 24 hours on Nov. 28th, 2001, including 20,945 valid connections. By *valid*, we mean that a socket connection was established and the handshake was successful. At the same time, 36,000 incoming requests were refused and 6,000 outgoing socket creations failed. The average duration for all sessions was 31 sec. and the average set-up time was 0.21 sec. It is difficult to reason about an *average* connection, however, because the distribution is highly skewed and results are predicated by a small number of very large values. In this case, the longest session lasted about 11 hours (39,973 seconds) and 5 sessions lasted over 8 hours. Table 2 (Experiment C) gives an indication of the distribution of connection duration.

In a more recent experiment, maintaining 5 connections over 1 hour on Dec. 30th, 2001, there were 297 valid Gnutella sessions for which the average set-up time was 1.05 sec. and the average duration was 57 seconds. Again the distribution was highly skewed and results are tabulated in Table 2 (Experiment D).

Table 2. Duration of Valid Connections

	Experiment C	*Experiment D*
Average	31 sec.	57 sec.
Median	0.17 sec.	0.4 sec.
Std. dev.	717 sec.	319 sec.
Max.	6350 sec.	3233 sec.
Average top 1 %:	2973 sec.	2960 sec.
Average top 10 %:	307 sec.	540 sec.
Average bottom 90 %:	0.26 sec.	2.3 sec.

The main conclusion is that the average duration of a connection is quite short, between 30 seconds and a minute.

5.6 "Good" Hosts

Having determined that the majority of Gnutella participants are transients who only connect to the network occasionally and then for short periods, we then set forth to see if the reliable hosts that we identified during one session could be reused in future sessions. If so, one could dispense with the need to connect to the same well-known host caches on start-up.

First, we extracted "good" connections from experiments done over 24 hours on December 30th, 2001. Our criterion for selection of a "good" host address was one to which the connection had remained active for at least 2 minutes (over twice the average connection duration). From 41,789 recorded connections, 564 connections (1.3 %) were considered "good."

Next, we scheduled periodic executions of the CTester program to see if it was still possible to re-establish socket connections to the "good" hosts. To obtain the public port for incoming connections, we send a Ping and waiting for a Pong with a hop count of 1. If they don't answer within a reasonable time, we assume the standard port 6346. Out of our 564 selected addresses, 191 (34 %) were incoming connections and of those only about a third (70) answered our Ping. In 75 % of these cases the public port returned was 6346; justifying our choice of that address for hosts that do not answer. Parenthetically, the fact that two thirds of our "good" hosts never responded to a Ping shows the difficulty in trying to measure network size by Pinging hosts!

The day after the addresses were obtained, we scheduled experimental runs every four hours over a 24 hour period. After this, we ran the experiment once a day for a week. During the first two days, the success rate for reconnection dropped steadily from about 18 % to 10 %. A week later, it reached 7 % where it has remained — varying between 6.4 % and 7.8 %.

This result may seem disappointing especially since in 380 cases (67 %) we were unable to reconnect even a single time. However, there were 4 hosts that we were always able to reach and another 57 who were available 50 % of the time or better. Additional experiments showed that we could open Gnutella sessions to 90 % of the hosts to which Ctester could open a socket. Thus it is possible to identify reliable semi-permanent Gnutella Hosts.

6 Measuring the Performance of Gnutella Applications

Beyond simply understanding the factors affecting the Gnutella network, our research is also aimed at improving the performance of applications. However, as demonstrated in our previous experiments, the performance of any one session depends on chance and measures of performance can therefore vary widely. Furthermore, the activity on the network varies with time. To be able to evaluate the effect of various servent parameters or strategies, we had to develop a methodology that would mitigate these problems.

The effect of random variation in performance can be reduced by running the servent over long periods, running multiple experiments at different times of the day and on different days of the week, and using averages from these runs.

However, it remains difficult to compare different algorithms. Clearly, we cannot compare two executions done at different time of the day or on different days, since there is no guaranty that the Gnutella network will be in the same state. Our solution is to run test programs in parallel with a fixed benchmark and to consider the performance *relative* to the benchmark.

Another basic problem is choice of a measure of performance. Over the course of our study, we used several indicators:

- the total number of messages,
- the total number of Pings,
- the total number of Pongs,
- the average horizon, and
- the number of distinct host addresses found.

No measure stood out as a best indicator. As a result we used them all and gave them equal weight. This yields the following experimental methodology:

- for each parameter value (or strategy) that we wish to test, we run an experiment which lasts 24 hours and consists of 24 runs (of 45 minutes), once every hour,
- for each run, we launch two (2) servents in parallel, the test servent and a benchmark servent whose parameters are constant for all experimental runs,
- on each run, for each servent, we record the values of the 5 indicators listed above,
- the statistics collected serve to compute the *performance ratio*, noted r, of the test servent:

$$r = \frac{1}{m} \cdot \sum_{i=1}^{m} \frac{\sum_{j=1}^{24} x_{ij}^{t}}{\sum_{j=1}^{24} x_{ij}^{b}}$$

  where
  - m is the number of indicators used,
  - x_{ij}^{s} is the value of indicator i collected at the j^{th} run of servent s, where s equals b for the benchmark servent and t for the test servent.

We conducted a preliminary evaluation of this methodology to assess how the targeted number of connections (K) influenced the performance of a servent. We expected that performance should increase with K and perhaps taper off for very large values as bandwidth limitations start to play a role. We ran a series of experiments where the test

servent used different values for the number of open connections. In particular, we had $K \in \{3, 4, 6, 8, 10, 15\}$ while the benchmark servant used a fixed value of five open connections ($K = 5$).

The results of the experiment are quite compelling and appear to be linear (Fig. 5) [4]. From this experiment, we observe that a servent with less than 2 target connections (1.68 to be exact) would not receive any traffic. It seems that 2 connections are always occupied trying to replace failed connections. Furthermore, there is no sign of tapering off; we could still increase performance by increasing K.

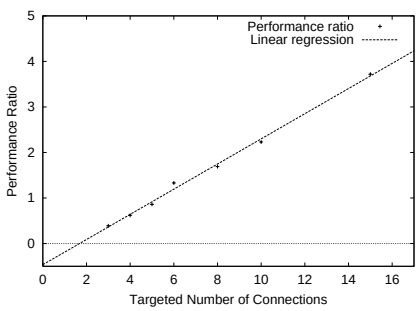

Fig. 5. Performance Ratio (r) vs. Targeted Number of Connections (K)

7 Conclusion

We have investigated the behaviour of participants in Gnutella, a well-known Internet community. Although some of the phenomena observed are particular to Gnutella, many of our results are relevant to other communities.

An important observation is the highly random nature of network behaviour. Repeatedly we observed a "whales and minnows" phenomenon whereby *average* measurements are determined by a small number of rare events with huge values and are therefore neither representative of the rare events nor of the more common small values. For example, we measured the average duration of a session to be 31 seconds, but 1 % of the sessions average 3000 sec. whereas the majority (99 %) average 1.3 sec. It is thus very difficult to get reproducible results.

Our experiments also showed that the composition of the community changes quite rapidly. Contrary to published results that suggest that connections last in the order of minutes or even hours [20], we found that sessions are much shorter: the duration of the average (median) session is less than half a second: 0.17 sec in one case and 0.4 sec in the other; and in another experiment we found that 98.7 % of the sessions lasted less than 2 seconds.

To maintain connectivity, nodes continuously exchange addresses of connected hosts that can be used to replace failed neighbours. A surprising observation was that a large

[4] $r = 0.276 \cdot (K - 1.68)$

proportion of the information thus obtained is incorrect or redundant: 80 to 90 % in the case of Gnutella's Pongs. A major part of the problem comes from hosts on subnetworks sending local (NAT) addresses which have no global validity. Even after filtering out flawed addresses, only a third of connection attempts result in a valid connection.

More generally, collaborative behaviour requires the exchange of organizational data between participants but flawed information may be a fact of life in open systems with unscreened participants, evolving technology, and a wide variety of software implementations.

Mapping the network or even estimating a *horizon* (the reachable portion of the net) may be more difficult than is generally believed. $2/3$ of our "good" hosts never acknowledged a Ping; other nodes do not forward Pings to their neighbours but return addresses from a local cache. The maximum horizon that we measured during our tests was less than 1000 nodes for 1 minute peaks. Average horizon values were much lower, normally in the hundreds.

We discovered semi-permanent reliable hosts but again they are rare. Starting from 42,000 site addresses we ended up with only 57 sites that were up 50 % of the time or better.

We developped an effective methodology — based on *comparative* measures and replication — to overcome the stochastic nature of network activity and allow the evaluation of various operating strategies.

In conclusion, the experimental investigation of Gnutella has revealed many interesting technical findings as well as conceptual insights. It became clear that a local intelligent screening and processing of community information is central for efficiency as well as scalability of such networks. Future work will thus concentrate on evaluating more sophisticated policies and strategies in both the real world of Gnutella and in simulated environments.

Acknowledgements

We would like to thank Cirano (Center for Inter-university Research and Analysis of Organizations, Montreal) where J. Vaucher is on sabbatical leave, for providing the machines and environment for this research.

Matthias Klonowski, an exchange student from Rostock, Germany, worked on the Jtella software during the summer of 2001. He pointed out several performance bottlenecks and helped discover a critical Java bug [22].

References

1. The gnutella protocol specification v0.4. Clip2 Distributed Search Services. Available from `http://www9.limewire.com/developer/gnutella\_protocol_0.4.pdf`
2. What is gnutella. LimeWire LLC, 2001.
   Available from `http://www.limewire.com/index.jsp/what_gnut`
3. Gnutella network size (realtime graph). LimeWire LLC, 2002. Available from `http://www.limewire.com/`

4. E. Bonabeau, G. Theraulaz, J. Deneubourg, S. Aron, and S. Camazine. Selforganization in social insects. *Trends in Ecol. Evol. 188–193*, 1997.
5. I. Clarke, O. Sandberg, B. Wiley, and T. Hong. Freenet: a distributed anonymous information storage and retrieval system. In *ICSI Workshop on Design Issues in Anonymity and Unobservability*, Berkeley, CA, 2000.
6. Gary Flake, Steve Lawrence, and C. Lee Giles. Efficient identification of web communities. In *Sixth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 150–160, Boston, MA, August 20–23 2000.
7. L. N. Foner. YENTA: A multi-agent referral system for matchmaking. In *First International Conference on Autonomous Agents*, Marina del Rey, California, 1997.
8. David Gibson, Jon M. Kleinberg, and Prabhakar Raghavan. Inferring web communities from link topology. In *UK Conference on Hypertext*, pages 225–234, 1998.
9. Francis Heylighen. Collective intelligence and its implementation on the web: Algorithms to develop a collective mental map. *Computational & Mathematical Organization Theory*, 5(3):253–280, 1999.
10. S. Ravi Kumar, Prabhakar Raghavan, Sridhar Rajagopalan, and Andrew Tomkins. Trawling the web for emerging cyber-communities. *WWW8 / Computer Networks*, 31(11-16):1481–1493, 1999.
11. K. McCrary. The gnutella file-sharing network and java. JavaWorld, 2000. Available from `http://www.javaworld.com/javaworld/jw-10-2000/jw-1006-fileshare.html`
12. K. McCrary. Jtella: a java api for gnutella, 2000. Available from `http://www.kenmccrary.com/jtella/index.html`
13. V. Menko, D. J. Neu, and Q. Shi. AntWorld: a collaborative web search tool. In Kropf et al., editor, *Distributed Communities on the Web (DCW)*. Springer Verlag Berlin, 2000.
14. Stanley Milgram. The small-world problem. *Psychology Today*, 1967.
15. S. Osokine. Gnutella blown away? not exactly, 2001. Available from `http://www.openp2p.com/pub/a/p2p/2001/07/11/numbers.html`
16. Y. Rekhter, B. Moskowitz, D. Karrenberg, G. J. de Groot, and E. Lear. Address allocation for private internets. Request for Comments 1918, Internet Engineering Task Force, February 1996. Available from ftp://ftp.isi.edu/in-notes/rfc1918.txt.
17. M. Ripeanu. Peer-to-peer architecture case study: Gnutella network. Computer Science Dept., University of Chicago, 2001. Available from `http://www.cs.uchicago.edu/files/tr\_authentic/TR-2001-26.pdf`
18. J. Ritter. Why gnutella can't scale. no, really, 2001. Available from `http://www.darkridge.com/~jpr5/doc/gnutella.html`
19. C. Rohrs. Limewire design. LimeWire LLC, 2001. Available from `http://www.limewire.org/project/www/design.html`
20. S. Saroiu, P. Gummadi, and S. Gribble. A measurement study of peer-to-peer file sharing systems. Technical report UW-CSE-01-06-02, University of Washington, 2002.
21. J. Sullivan. Napster: Music is for sharing. Wired News, November 1999. Available from `http://www.wired.com/news/print/0,1294,32151,00.html`
22. J. Vaucher and M. Klonowski. Bug #4508232: Unborn threads are not garbage collected. Sun Developper Connection, 2001. Available from `http://developer.java.sun.com/developer/bugParade/bugs/4508232.html`