Bootstrapping in Gnutella: A Preliminary Measurement Study

Pradnya Karbhari, Mostafa Ammar, Amogh Dhamdhere, Himanshu Raj, George Riley, Ellen Zegura
E-mail: {pradnya@cc, ammar@cc, amogh@cc, rhim@cc, riley@ece, ewz@cc}.gatech.edu

College of Computing Tech Report Number GIT-CC-03-35
Georgia Institute of Technology, Atlanta, GA-30332
May 2003

***Abstract*—**
To join an unstructured peer-to-peer network like Gnutella, peers have to execute a *bootstrapping function* in which they discover other on-line peers and connect to them. Until this bootstrapping step is complete, a peer cannot participate in file sharing activities. Once bootstrapping is complete, a peer's experience is strongly influenced by the choice of neighbor peers resulting from the bootstrapping step. Despite its importance, there has been very little attention devoted to understanding the behavior of this bootstrapping function. In this paper, we study the bootstrapping process of a peer in the Gnutella network. This is a preliminary investigation, consisting of 1) an analysis and performance comparison of bootstrapping algorithms of four Gnutella servent implementations, 2) a measurement-based characterization of the global Gnutella Web Caching System (GWebCaches), a primary component of the current bootstrapping functions, and 3) a study of the behavior and experience of a single GWebCache that was setup locally and made part of the global caching infrastructure. Our study highlights the importance of understanding the performance of the bootstrapping function as an integral part of a peer-to-peer system. We find that 1) there is considerable variation among various servent implementations that correlates to their bootstrapping performance, 2) even though the GWebCache system is designed to operate as a truly distributed system in keeping with the peer-to-peer system philosophy, it actually operates more like a centralized infrastructure function, and 3) the GWebCache system is subject to misreporting of peer and cache availability due to stale data and absence of validity checks.

I. Introduction

To join an unstructured peer-to-peer network like Gnutella, peers have to execute a *bootstrapping function* in which they discover other on-line peers and connect to them. These initial neighbor peers determine the new peer's location in the overall Gnutella topology, and ultimately its search and download performance. Also, from the user perspective, the time spent by the peer in bootstrapping is critical because until the bootstrapping step is complete, a peer cannot participate in file sharing activities such as searching and downloading. From our experience, this time can vary significantly for different Gnutella servents[1].

This work was supported by NSF grant ANI-9973115 and the Georgia Tech Broadband Institute.

[1]The implementations of Gnutella peers are referred to as *servents* because they function as *serv*ers and as cli*ents*. We use the terms *peers* and *servents* interchangably.

Despite the significance of the bootstrapping process in unstructured peer-to-peer networks, it has received very little attention to date. There have been various studies[1], [2] aimed at characterization of peers based on their uptimes, bottleneck bandwidths, latencies and other factors, and trying to improve a peer's search and download experience[3]. None of these however have studied the bootstrapping function.

Initially Gnutella users relied on word of mouth to determine the address of an on-line peer that would allow newly joining peers to tap into the network. The use of automated caching servers as well as caching in the Gnutella servent itself, was introduced at a later time. As Gnutella gained in popularity after Napster was shut down, the caches ultimately became the pre-dominant bootstrapping technique [4]. Anecdotally, it has been observed that the switch from the use of word of mouth to the use of automated caches resulted in a signifcant change to the structure of the Gnutella network and a worsening of its performance[4].

In this paper we undertake a measurement study of the current bootstrapping process in the Gnutella network. Our investigation consists of three parts—

1) An analysis and performance comparison of the bootstrapping algorithms of four Gnutella servent implementations: LimeWire[5], Mutella[6], Gtk-Gnutella[7] and Gnucleus[8].
2) A measurement-based characterization of the global Gnutella Web Caching System[9] (GWebCaches), a primary component of these bootstrapping algorithms. The GWebCache system is a network of caches, each of which maintains a list of other caches in the system and a list of online hosts that are accepting incoming connections. Peers that want to join the network can send a request to one of these caches and retrieve the host list.
3) A study of the behavior and experience of a single GWebCache that was set up locally and made part of the global caching infrastructure.

Based on our analysis of the data collected, we highlight below our three main findings about the current Gnutella bootstrapping system. These are just preliminary results, and we intend to continue the analysis based on more current data.

- Although similar in the basic structure of the algorithm and the data structures used, the servent implementations differ in the details, with significant impact on their bootstrapping times, as seen in our measurements.

- An analysis of the request rates at different caches points to the disparity in traffic volume handled by these caches– some caches are very busy, and their host and cache lists evolve much faster than some others. The load balancing goal of any distributed system is not really achieved in this system, thus making the system operate more as a centralized infrastructure.
- The GWebCache system is subject to misreporting of peer and cache availability. This is because the data reported in the updates to these caches is not subjected to any validity checks by the caches. Peers might thus waste time trying to connect to off-line hosts returned by the GWebCaches.

The rest of the paper is structured as follows. In Section II, we give an overview of the bootstrapping process in different Gnutella servents, with special focus on the GWebCache system. In Section III we discuss the performance of the different servents with respect to their bootstrapping times. In Section IV we discuss the performance of the GWebCache system. In Section V, we summarize our findings and discuss future work.

II. Gnutella Bootstrapping

A peer intending to join the Gnutella network needs to know the addresses of online peers in the network. Currently, the GWebCache system functions as a distributed repository for maintaining this information. Peers can query the caches in this system to get a list of online peers, and try connecting to them. In the first run of a particular Gnutella servent, the GWebCache system is the only means available to the servent to locate other online peers. In successive runs, individual servent implementations might try other approaches (apart from the GWebCaches), such as maintaining local lists of hosts seen during their earlier runs. We first discuss the GWebCache system, as it is an important component of the bootstrapping functionality, and is essential in the understanding of the servent bootstrapping algorithms.

A. Gnutella Web Caching System

The GWebCache system[9] is a network of caches that maintain a list of online peers accepting incoming connections. When a new peer wants to join the Gnutella network, it can retrieve the host list from one or more of these GWebCaches. The GWebCaches also maintain a list of other caches in the system.

The GWebCaches are voluntarily operated. When a new cache wants to join the system, a servent needs to explicitly advertise the address of the cache to other caches. These caches then include this new cache in their cache list. Typically each cache maintains a list of 10 other caches and 20 hosts which are currently accepting incoming connections.

The peers in the Gnutella network are responsible for keeping the information in these caches up-to-date; the caches do not communicate with each other at any time. A host accepting incoming connections is supposed to update the caches with its IP address and port number. As a security feature, a cache accepts update requests from the same IP address a maximum of once every hour. The host and cache lists maintained at the caches are first-in-first-out lists. Hence, as new hosts update a cache with their addresses, old hosts are removed from the list. When a host updates a cache with its IP address, it is supposed to also update the cache with information about some other cache that it believes is alive. Again, the cache list gets updated with more recent information about live caches.

TABLE I
GWebCache messages

argument	cache response
$ping$=1	$pong$ message to servent
$urlfile$=1	list of caches
$hostfile$=1	list of online hosts
ip=<IPaddress>	host list is updated with IP address
url=<URL of cache>	cache list is updated with URL
$statfile$=1	access statistics over last hour

Table I lists the messages sent by a client using the GWebCache protocol. A simple HTTP request, in the form– "URL?$argument$" is sent to the webserver at which the cache is located. The caches respond as shown in the table. Note that the GWebCaches do not maintain any information about the online hosts, other than their IP addresses and port numbers.

B. Servent Bootstrapping Algorithms

In this section, we discuss the bootstrapping algorithms of the Gnutella servents that we compared, and point out the differences between them. We analyzed Limewire v2.9[5], Gtk-Gnutella v0.91.1[7], Mutella v0.4.3[6] and Gnucleus v1.8.6.0[8]. All these versions support retrieval from and updates to the GWebCache system. The bootstrapping processes in the four servents are similar in their use of the GWebCache system and the local caching of hosts.

The data structures maintained by these servents include a list of known caches which is periodically populated with the addresses of new caches. Servents also maintain lists of *known* hosts and *permanent* hosts, the definitions of which differ slightly in different servents. Informally, permanent hosts are hosts that the servent managed to contact in current and previous runs. Some servents also maintain a list of ultrapeers[2].

The generic bootstrapping algorithm for the servents is as follows:

1) Initialize the following data structures in memory by reading the corresponding files from disk—
    - list of caches
    - list of known hosts
    - list of permanent hosts
    - list of ultrapeer hosts (not in Gtk-Gnutella)
2) Depending on mode (ultrapeer/normal), determine the minimum number of connections to be maintained.
3) Try to establish the minimum number of connections to peers in the following order:
    - In LimeWire and Gnucleus, try to connect to ultrapeers.

[2]Ultrapeers[10] are hosts which have higher bandwidth and CPU power, sufficiently long uptime and are not firewalled. Normal or leaf nodes have low CPU and bandwidth capabilities and typically connect to ultrapeers.

TABLE II

SERVENT IMPLEMENTATION DIFFERENCES

Characteristic	LimeWire	Mutella	Gtk-Gnutella	Gnucleus
Maintains ultrapeers list?	Yes	Yes	No	Yes
Priority to ultrapeers when connecting?	Yes	No	No	Yes
Host and cache lists prioritized by age?	Yes	No	Yes	No
Updates to GWebCaches	Ultrapeer mode	Ultrapeer mode	Any mode	Any mode
Number of hardcoded caches	181	3	3	2

- Try to connect to any host in the known hosts and permanent hosts lists.
- If the servent is still not connected, request the host list from a GWebCache (multiple GWebCaches in case of LimeWire) and try to connect to these hosts.

4) Periodically, a connection watchdog checks whether the minimum number of connections (from step 2) are alive. If not, try to establish a new connection as stated above.
5) Periodically update a cache with its own IP address and URL of another cache (for LimeWire and Mutella, this is done only if in ultrapeer mode)
6) On shutdown, write the different files to disk, for retrieval on next startup.

We now highlight the differences in the four servent implementations, shown in Table II.

- Limewire and Gnucleus maintain a separate list for ultrapeers and give priority to hosts in this list during connection initiation. Since ultrapeers have relatively long uptimes and the capability to support more incoming connections, prioritizing these peers during connection initiation increases the chances of successfully connecting to a peer. Although Mutella maintains a list of ultrapeers separately, this information is not used during bootstrapping. When establishing a connection, it randomly picks a host from the lists of ultrapeers, known hosts and permanent hosts. Gtk-Gnutella does not distinguish between ultrapeers and normal peers, thus performing relatively poorly in terms of its bootstrapping time.
- LimeWire and Gtk-Gnutella prioritize their host and cache lists by age. This enables them to act on more recent information.
- Although all four servents we examined support the GWebCache system for retrieving information, LimeWire and Mutella support updates only in the ultrapeer mode. This is better for the system because the probability of ultrapeers accepting incoming connections is higher. Gtk-Gnutella and Gnucleus update the GWebCaches even in the leaf mode (when their limit on number of connections is low), and hence might not be accepting incoming connections by the time a peer tries to connect to them.
- The Gnutella servents have a set of hardcoded caches, which are used during the very first run of the servent, before any other information about caches or hosts is known. The number of hardcoded caches in the servents are shown in the table. LimeWire has a surprisingly high number of hardcoded caches (181), out of which 135 caches were active when we tried to ping them at the Gnutella level.

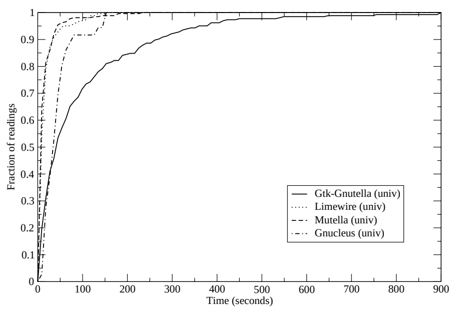

Fig. 1. CDF of bootstrapping times of servents at university

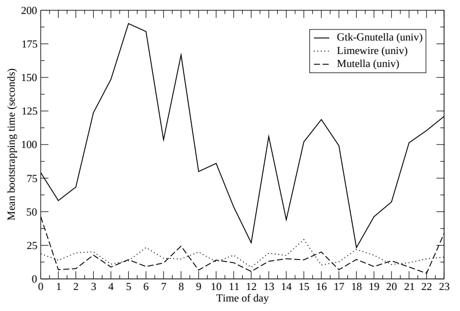

Fig. 2. Mean bootstrapping times at different times of the day

In the next section, we will discuss the effects of these differences on the performance of different servent implementations.

III. BOOTSTRAPPING MEASUREMENT AT SERVENT

In this section, we compare the performance of the servents considered in our study, based on their *bootstrapping times*. We define the bootstrapping time of a servent as the time between the start of the servent and the establishment of the first *stable* Gnutella-level connection[3]. We say that a connection is *stable* if it is maintained for at least $threshold$ seconds.

A. Measurement Methodology

We modified the source code of the three Linux-based servents (LimeWire, Gtk-Gnutella and Mutella) to log the times at which the application was started and shut down. We also

[3]A "Gnutella-level" connection is established after the Gnutella handshake messages are exchanged between the two connecting peers.

logged the time when a Gnutella-level connection was established and terminated. For the Windows-based servent (Gnucleus), we used Windump[11] to collect packet traces and then determined the connection times by analyzing the traces.

We started the Linux-based servents once every hour, synchronously at two locations— at a university campus on a Fast Ethernet Link and at home on a DSL link to a local ISP. We started Gnucleus once every three hours at the university location only. Each servent was allowed to run for 15 minutes, after which it was shut down. In the following section we analyze the bootstrapping times measured during an 11-day experiment.

A limitation of our study is that both the locations in our experiments are high bandwidth links. We did not run any experiments at a slower access link.

B. Performance Measurements

Figure 1 shows the cummulative distribution function of the bootstrapping times of the four servents at the university location. In this graph we set $threshold$ to 120 seconds. We analyzed the bootstrapping times with different values for $threshold$ and observed similar results. The graphs for the bootstrapping times of servents on the DSL link are also similar.

The most striking observation is that Gtk-Gnutella performs much worse than Mutella and LimeWire. We conjecture that this is due to the fact that Gtk-Gnutella does not differentiate between ultrapeers and normal peers. Also, once it selects a particular cache to contact for retrieving the host list, it sticks to it for 8 consecutive updates or retrievals. In Section IV, we will see that cache quality varies, hence maintaining a poor choice of cache can have a significant effect. Gnucleus also performs worse than Mutella and LimeWire, but better than Gtk-Gnutella. This is probably because the GWebCache list and the different host lists are not prioritized by age in the Gnucleus implementation.

Figure 2 shows the mean bootstrapping times for the three Linux-based servents at the university location for different times of the day. LimeWire and Mutella perform almost the same throughout the day. Gtk-Gnutella, which does not differentiate between ultrapeers and normal peers performs similar to LimeWire and Mutella when there is a large number of normal peers online in the system (around noon or late afternoon). When there are very few normal peers around (early in the morning), Gtk-Gnutella shows a higher mean bootstrapping time. This highlights the importance of ultrapeer awareness on the part of a Gnutella servent.

Since we started multiple instances of Gnutella servents on the same local area network, we observed how many of the experiments discovered and connected to our own servents. In spite of our expectations to the contrary, not a single experiment out of 264 experiments over 11 days was able to discover the nearby peers. This highlights the lack of Internet location awareness in the GWebCache system and in the local host list of the servents. We were expecting Gtk-Gnutella to show some local network connections. This is because in its connection setup algorithm, when it discovers a peer on the local network (through a $pong$ message), it prefers to connect to that peer and disconnect from some other existing peer. The absence of such connections indicates that the Gtk-Gnutella servent either did not find the other three servents on the Gnutella network, or it found one of them, but was unable to establish a connection with it, as the other servent was not accepting any incoming connections.

The performance of the GWebCache system has a significant impact on the bootstrapping times of servents seen above. We therefore analyze the performance of this system in the next section.

IV. GWebCache Performance

In order to gain a better understanding of the GWebCache system, we performed a measurement study of the system at two levels We conducted a global study by periodically crawling all the caches in the GWebCache system and retrieving information from them. We conducted a local study by setting up our own cache and analyzing its access patterns.

A. Global GWebCache System Performance

Through analysis of the GWebCache system, our goal is to answer the following questions:

1) How many caches are active in the system at any time? What does the evolution of the cache list at a single cache and at all caches look like?
2) What are the access patterns for different requests (cache list, host list, and updates) at different caches? What are the differences in access patterns across different caches and in the whole system?
3) What does the evolution of the host list at a single cache and at all caches look like? What percentage of the new hosts added to the caches at any time are unique hosts?
4) In the host list returned by the caches, how many hosts are actually alive, how many of them are accepting Gnutella-level connections, and how many of them are ultrapeers?

1) Measurement Methodology: We studied the system by polling the caches at regular intervals. Requests in the format shown in Table I were sent to each of the caches, according to the information required. We collected multiple traces over a one month period.

In order to answer the first question, we retrieved the *cache list* every 30 minutes starting with a seed cache and crawled the caches returned, until we had polled all the caches in the system. We also determined the number of active caches in the system by sending Gnutella $ping$ messages to these caches.

To answer the second question, we retrieved the *statistics file* every hour from each active cache. The statistics file gives the number of update requests and total requests the cache received within the last hour. The total requests include the requests for the cache list, the host list and update requests for both lists.

In order to answer the third question, we retrieved the *host list* from the active caches every 5 minutes, and studied its evolution at a particular cache and in the whole system.

To answer the fourth question, we sent Gnutella-level connect messages to the hosts in the host lists returned by the caches. If we managed to establish a TCP connection, we determined that the host was alive. If we established a Gnutella-level connection, we determined that the host was accepting incoming connections. Out of the hosts that responded with the

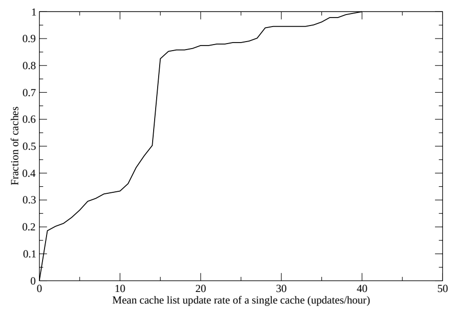

Fig. 3. CDF of mean cache list update rate at a single cache

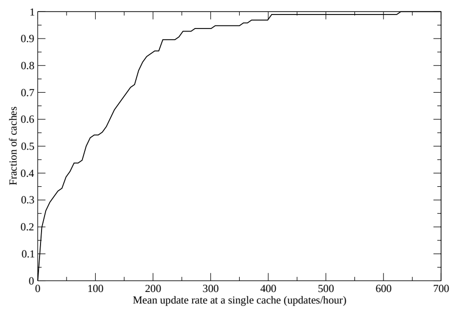

Fig. 5. CDF of mean update rate at a single cache

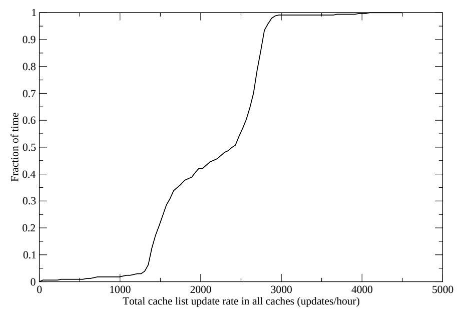

Fig. 4. CDF of total cache list update rate in all caches

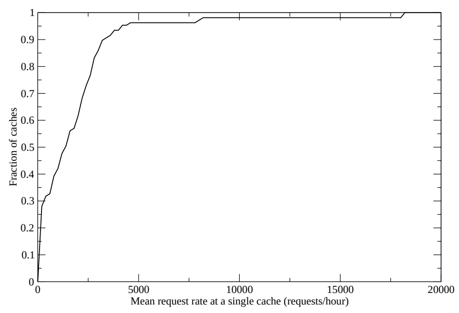

Fig. 6. CDF of mean request rate at a single cache

proper $pong$ response, we determined whether the host was an ultrapeer or not, using a field *X-Ultrapeer: True/False* in the response.

These methods have several limitations. Since we polled the caches starting with a seed cache, we will miss caches in any disconnected components of the GWebCache system. Also, between the time we retrieved the list and the time we actually tried to connect to the peer, the peer could have gone offline. We assume that the information returned by the caches during any of our polls is valid (i.e., the caches are not misconfigured or misbehaving).

2) Analysis of the cache list: Although we located 523 caches over the period of the study (15 days), fewer than half of them–222– responded to our requests. During any particular polling period, at most 150 caches were active. The others were either unreachable or did not respond with the correct data. This is a surprisingly low number of reachable caches, considering the dependance on these caches for bootstrapping purposes.

Figure 3 shows the CDF of the mean cache list update rate at a single cache. Notice that some of the caches are very active, with update rates of about one cache per minute, whereas other caches have very low update rates. About 40% of the caches have an update rate of about 15 per hour, and about 80% have an update rate of 15 per hour or less. This indicates that the system is not as distributed as we would expect it to be. The fact that all the servents we studied have some hardcoded caches, indicates that the request rates to these caches could be very high.

Figure 4 shows the CDF of the cache list update rate seen by all caches in the entire system during a particular poll. Most of the update rates (about 90%) are between 1300 and 2800 updates per hour, with two noticeable modes at about 1350 and 2800 updates per hour.

3) Analysis of access patterns: An analysis of the request rates and the evolution of the host and cache lists points to the disparity in the type of caches in the system. Some caches are very busy, and their lists evolve much faster than some others, as we will see in the discussion below.

Figure 5 shows the CDF of the mean (host and cache) update rates at a single cache. About 40% of the caches receive update rates of 50 per hour or less, about 80% of the caches get update rates of 200 per hour or less, and there are a few caches with very high update rates. Similarly, Figure 6 shows the CDF of the mean total request rates at a single cache. These include re-

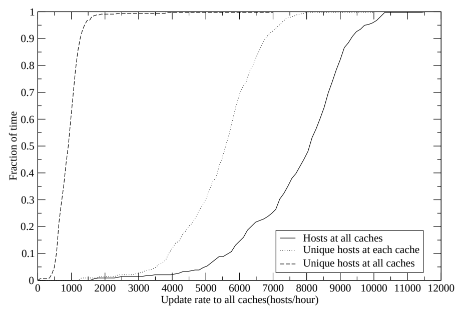

Fig. 7. Host update rates in all caches

quests for the host and cache lists and updates to both lists from peers. About 50% of the caches receive a request load of 1000 per hour, whereas there are some caches that receive extremely high loads— on the order of 10000 requests per hour. This shows that some caches can potentially be extremely stressed.

4) Evolution of the host list in caches: As expected, the host list evolves much faster than the cache list in any cache. During a 15-day period in our study, we saw over 300000 unique IP address:port combinations in all caches.

Figure 7 shows the CDF of the host update rates at all caches in the system. The rightmost line shows the CDF of the host updates received at all caches in the system. The dotted line at the center shows the CDF of the host updates with unique IP address:port combination at each cache. The leftmost curve with the dashed line shows the CDF of the unique IP address:port combination seen in the whole system. The average rate for unique IP address:port updates is significantly lower than the actual update rate. Thus, the same hosts (presumably ultrapeers) keep updating the caches frequently with their IP addresses.

5) Analysis of host behavior in host lists: When we tried connecting to the hosts in the host lists retrieved, on an average we found 50% peers online, 16% peers accepting incoming Gnutella-level connections, and 14% ultrapeers. This shows that a surprisingly low number of peers indicated in the caches are actually accepting incoming connections. This might be because of the high timeout enforced (55 minutes) by the caches. During this period, a peer might have either established its minimum number of connections or might have gone offline.

B. View of a local cache

In this section, we look at the view of a single cache in the GWebCache system. We set up a cache locally using a PHP script for the GWebCache v0.7.5 and advertised it to the global caching infrastructure.

We introduced the cache into the system using a feature in the Gnucleus servent. The servent first sends a Gnutella ping to the cache to ensure that it is alive. If the cache returns a valid *pong* message, then the Gnucleus servent iterates through its list of known caches, and sends an update request to each cache, with the URL of the new cache. The servent then adds the new cache to its local list of caches. We found that 17 caches pointed to our cache, immediately after the cache advertisement, after which the number stabilized to about 4 or 5 caches pointing to ours every hour.

Figure 8 shows the CDF of requests for the host and cache lists at the local cache. Our cache gets a request rate of about 15-20 per hour for the host list and about 5-10 per hour for the cache list. Figure 9 shows the updates received at the local cache every hour for a 7-day period. The black portion shows the number of cache updates, and the white portion shows the number of host updates at the cache in that hour. Comparing these request rates to the request rates of other caches, seen earlier in the section, we can see that our local cache is used less frequently than the other caches.

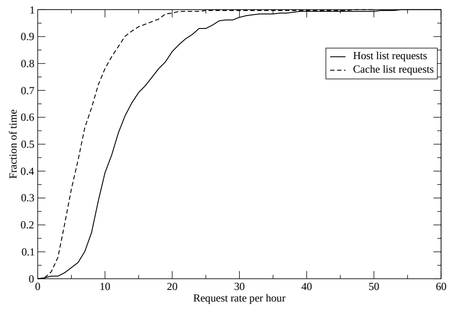

Fig. 8. CDF of requests for host and cache lists at local cache

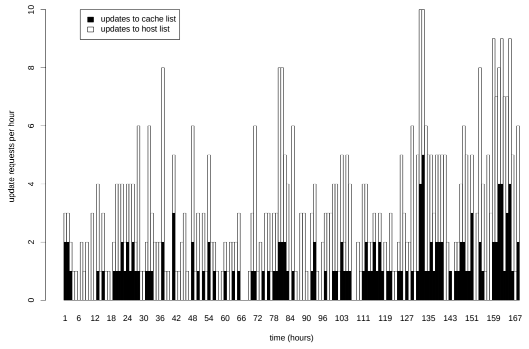

Fig. 9. Updates to host and cache lists at local cache

V. Conclusions

In this paper, we examined the bootstrapping function in unstructured peer-to-peer networks like Gnutella, wherein peers that want to join the network try to locate online hosts to peer with. These initial neighbors, determined by the bootstrapping function, strongly influence the search and download experience of a peer. Our study highlights the importance of understanding the performance of the bootstrapping function as an integral part of a peer-to-peer system.

Our examination of bootstrapping implementations in Gnutella servents shows considerable variation that correlates with their bootstrapping performance. We also investigated the GWebCache system which reveals its lack of true distributedness as well as its susceptibility to misreporting of peer and cache availability.

Our goal in performing this measurement study was to analyze and understand the functioning of the current bootstrapping system. In this study, we have done some preliminary analysis of the effect of the current bootstrapping system on the bootstrapping time of the peer. We further aim to analyze the effect of bootstrapping on the search and download experience of peers, and on the evolution of the Gnutella topology. These studies will lead to our ultimate goal of improving the bootstrapping process in unstructured peer-to-peer networks like Gnutella, by either suggesting improvements to the GWebCache system or proposing a new distributed bootstrapping system that will overcome the shortcomings of the current system. Bootstrapping is also an important function in structured peer-to-peer networks such as Chord[12] and CAN[13]. We plan to investigate these in the future, although the concerns in these systems will be different.

References

[1] S. Saroiu, P. Gummadi, and S. Gribble, "A measurement study of peer-to-peer file sharing systems," in *Proceedings of Multimedia Computing and Networking*, 2002.

[2] J. Chu, K. Labonte, and B. Levine, "Availability and locality measurements of peer-to-peer file systems," 2002.

[3] T.S. Eugene Ng, Y. Chu, S. Rao, K. Sripanidkulchai, and H. Zhang, "Measurement-based optimization techniques for bandwidth-demanding peer-to-peer systems," in *Proceedings of IEEE Infocom 2003*.

[4] Andy Oram, *Peer-To-Peer*, O'Reilly, 2001.

[5] "LimeWire," http://www.limewire.com.

[6] "Mutella," http://mutella.sourceforge.net/.

[7] "Gtk-Gnutella," http://gtk-gnutella.sourceforge.net/.

[8] "Gnucleus," http://www.gnucleus.net/.

[9] "Gnutella Web Caching System," http://www.gnucleus.net/gwebcache/.

[10] "Ultrapeer Specifications," http://www.limewire.com/developer/Ultrapeers.html.

[11] "Windump," http://windump.polito.it/.

[12] I. Stoica, Robert M., D. Karger, M. F. Kaashoek, and H. Balakrishnan, "Chord: A scalable peer-to-peer lookup service for internet applications," in *Proceedings of ACM Sigcomm*, 2001.

[13] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker, "A scalable content addressable network," in *Proceedings of ACM Sigcomm*, 2001.