Bootstrapping in Gnutella: A Measurement Study

Pradnya Karbhari, Mostafa Ammar, Amogh Dhamdhere, Himanshu Raj, George Riley, Ellen Zegura

Georgia Institute of Technology, Atlanta, GA-30332
{pradnya@cc, ammar@cc, amogh@cc, rhim@cc, riley@ece, ewz@cc}.gatech.edu ⋆

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 completed, a peer's search and download 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. We find that (1) there is considerable variation among various servent implementations, and hence in their bootstrapping performance. (2) The neighbors of a peer, which are the outcome of the bootstrapping process, play a very important role in the peer's search and download performance. (3) Even though the GWebCache system for locating peers is designed to operate as a truly distributed caching system, it actually operates more like a centralized infrastructure function, with significant load imbalance. (4) The GWebCache system is subject to significant misreporting of peer and GWebCache availability, due to stale data and absence of validity checks.

1 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].

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,

⋆ This work is supported in part by NSF grants ANI-0240485 and ANI-9973115.

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* interchangeably.

bottleneck bandwidths, latencies and other factors, and trying to improve a peer's search and download experience[3]. However, none of these 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 significant 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 Gnutella Web Caching System[9] (GWebCaches), a primary component of bootstrapping algorithms.
3. A measurement-based analysis of the role of neighbor peers, resulting from different bootstrapping algorithms, in the search performance of a peer.

Based on our analysis of the data collected, we highlight below our four main findings about the current Gnutella bootstrapping system.

1. 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.
2. The neighbors of a peer play an important role in the search performance of the peer, thus pointing to the importance of the bootstrapping process.
3. An analysis of the request rates at different GWebCaches 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.
4. The GWebCache system is subject to significant misreporting of peer and cache availability. This is because the data reported in the updates to these caches is not validated by the caches.

The rest of the paper is structured as follows. In Section 2, we give an overview of the bootstrapping process in different Gnutella servents, with special focus on the GWebCache system. We discuss the performance of the different servents with respect to their bootstrapping times in Section 3, and the role of the resulting neighbor peers in the search performance, in Section 4. In Section 5 we discuss the performance of the GWebCache system. In Section 6, we summarize our findings and discuss future work.

2 Gnutella Bootstrapping

In this section, we describe the bootstrapping process in the Gnutella servents we analyzed, and the functioning of the GWebCache system.

Table 1. 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 and port number
url=<URL of cache>	cache list is updated with URL
statfile=1	access statistics over last hour

2.1 Gnutella Web Caching System

A peer intending to join the Gnutella network requires the IP 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 to connect to. In the first execution of a particular Gnutella servent, the only means to locate other online peers is the GWebCache system. In successive executions, individual servent implementations try 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.

The GWebCache system[9] is a network of voluntarily-operated 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. Typically each cache maintains a list of 10 other caches and 20 hosts that 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, and with information about some other GWebCache that it believes is alive. The GWebCaches maintain the host and cache lists as first-in-first-out lists.

Table 1 lists the messages sent by a client using the GWebCache protocol. An HTTP request of 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.

2.2 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 GWebCaches, which is periodically populated with addresses of new GWeb-

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 and list of ultrapeer hosts (except 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 order:
   - In LimeWire and Gnucleus, try to connect to ultrapeers.
   - 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 LimeWire) and try to connect to these hosts.
4. Periodically, a connection watchdog checks whether the minimum num of connections (step 2) are alive. If not, try to establish a new connection (step 3).
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.

Fig. 1. Generic bootstrapping algorithm

Table 2. Servent implementation differences

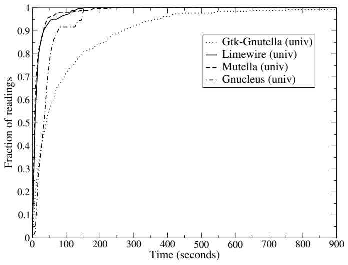
Characteristic	Limewire	Mutella	Gtk-gnutella	Gnucleus
Maintains ultrapeers list?	Yes	Yes	No	Yes
Prioritize ultrapeers when connecting?	Yes	No	No	Yes
Host & cache lists prioritized by age?	Yes	No	Yes	No
Updates to GWebCaches	Ultrapeer mode	Ultrapeer mode	Any mode	Any mode
Number of hardcoded caches	181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

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 separate list of ultrapeers[2].

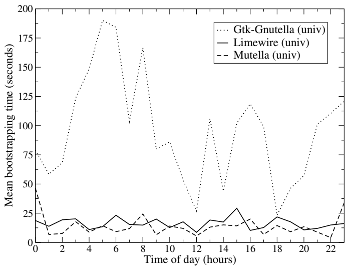
Figure 1 outlines the generic bootstrapping algorithm, and Table 2 summarizes the differences in the implementations, as discussed below.

1. Limewire and Gnucleus maintain a list of 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 probability of successfully connecting to a peer. Although Mutella also maintains a list of ultrapeers, this information is not used during bootstrapping. Gtk-Gnutella does not distinguish between ultrapeers and normal peers.
2. LimeWire and Gtk-Gnutella prioritize their host and cache lists by age. This enables them to act on more recent (and hence more accurate) information.
3. Although all four servents we examined support the GWebCache system for retrieving information, LimeWire and Mutella support updates to the GWebCaches only in the ultrapeer mode. This is better for the system because the probability of ultrapeers accepting incoming connections is higher than

[2] Ultrapeers[10] are hosts with high bandwidth and CPU power, and long uptimes. Normal or leaf nodes have lower capabilities and typically connect to ultrapeers.



(a) CDF of servent bootstrapping times



(b) Mean bootstrapping times acc. to time of day

Fig. 2. Servent bootstrapping times

when in the leaf mode. Gtk-Gnutella and Gnucleus update the GWebCaches even in the leaf mode.

4. 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. As seen in Table 2, compared to other servents LimeWire has a very high number of hardcoded caches (181), 135 of which were active when we tried to ping them at the Gnutella level.

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

3 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 initial 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.

3.1 Measurement Methodology

We modified the sourcecode of the three Linux-based servents (LimeWire, Gtk-Gnutella and Mutella) to log the times at which the application was started and shut down, and 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 analyzed them to determine the connection times.

We started the Linux-based servents once every hour, synchronously at two locations, at a university campus on a Fast Ethernet Link and at a residence 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. One limitation of our study is that

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

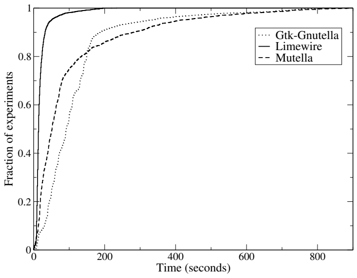


Fig. 3. Time to receive first queryhit

both locations in our experiments have high bandwidth access links. We did not run any experiments at a slower access link.

3.2 Performance Measurements

Figure 2(a) shows the cumulative 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 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 GWebCache to contact for retrieving the host list, it uses it for 8 consecutive updates or retrievals. In Section 5, we will see that cache quality varies; thus, maintaining a poor choice of cache can affect performance. 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.

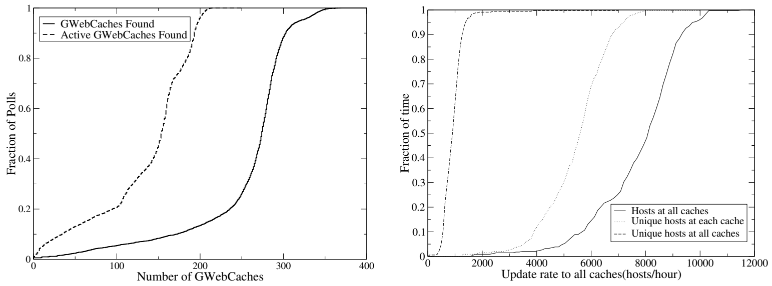
Figure 2(b) shows the mean bootstrapping times for the three Linux-based servents at the university location for different times of the day. LimeWire and Mutella perform almost the same throughout the day. Gtk-Gnutella, which does not differentiate between ultrapeers and normal peers performs similar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

Although we started multiple instances of Gnutella servents on the same local area network, none of our peers were able to discover each other in any one of our experiments over two weeks. This highlights the lack of Internet location awareness in the GWebCache system and in the local host list of the servents.

In the next section, we discuss the importance of neighbor peers (resulting from the bootstrapping process) in the search performance of peers.

4 Importance of Neighbor Peers

A peer gets access to the peer-to-peer network through its directly connected neighbors. The peers that these neighbors have access to, within an N-hop ra-



(a) Percentage of active caches (b) CDF of hostlist update rate

Fig. 4. Cache and host list update rates in all GWebCaches

dius (usually N=7), comprise the *neighborhood* of the peer. All query messages originated by the peer will be forwarded to this neighborhood. The number of peers in this neighborhood, the types of files shared, the number of files shared amongst all these peers will reflect on the search performance of a peer.

We studied the effect of neighbors on search performance of LimeWire, Mutella and Gtk-Gnutella for the the 15 most popular search queries[2]. The performance metric we considered is the time to get the first response, which is the time-lag from when the servent is bootstrapped and issues the query, to the time when it gets the first response. Figure 3 shows the CDF of this response time for the top 15 queries issued by that servent during any experiment.

We found that there is usually a significant variation in the time to get the initial response. Limewire performs the best, primarily because during bootstrapping it prioritizes ultrapeers, who usually have access to a larger neighborhood. Mutella and Gtk-Gnutella perform worse, and take more than 5 minutes to give a result, in about 10% of the experiments.

We conclude that for a good search experience it is very important to have a set of good neighbors that provide access to many peers, sharing many files.

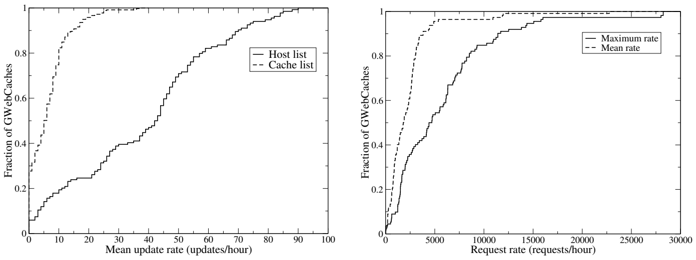
5 GWebCache Performance

We analyzed the performance of the GWebCache system with reference to the properties of a good distributed caching infrastructure (e.g., sufficient total system capacity, good load balancing, reliability of cached items, and physical or topological proximity of cached items served). With this goal in mind, we performed a measurement study of the system at two levels, globally and locally.

5.1 Global GWebCache System Performance

We studied the global GWebCache system by periodically crawling all the caches. We sent requests in the format shown in Table 1 to each active cache, according to the information required. We collected multiple traces over a five month period (Apr-Sept 2003), with the goal of answering the following questions.

1. How many GWebCaches does the system comprise of? How many of the reported caches are active at any time?



(a) CDF of update rates (b) CDF of request rates

Fig. 5. Update and Request rates at a single GWebCache

We retrieved the *cache list* every 30 minutes, starting with a seed GWebCache and crawled the caches returned, until we had polled all reachable caches. We also determined the number of active GWebCaches by sending Gnutella *ping* messages to the crawled list.

Although we found URLs of 1638 unique GWebCaches in 5 months, only one-fourth of them (403) were active at any time, and at most 220 of them were active during a single poll. This is quite a low number of reachable GWebCaches, potentially serving about 100000 active hosts[4] at any time on the Gnutella network, only 10% of which accept incoming connections. This indicates that the GWebCache system might get overloaded.

Figure 4(a) shows the CDF of the number of GWebCaches found during each of our polls, and the number of caches which were actually active (i.e. responded to out Gnutella-level *ping* messages). Most of the time, only about 160 caches out of 280, or about 60% were active. This is because the GWebCache system does not have any means of deleting an introduced cache. Since peers update caches with URLs of caches they know of (probably even cached from previous runs), without necessarily knowing whether they are alive or not, it is quite likely that inactive caches are reported for a long time.

2. What are the access patterns for different requests (cache list, host list, and updates) at different GWebCaches? What are the differences in access patterns across different GWebCaches and in the whole system?

We retrieved the *statistics file* every hour from each active GWebCache. The statistics file gives the number of update requests and total requests for cache and host lists the GWebCache received within the last hour.

Figure 5(a) shows the CDF of the mean update rates to the cache and host lists (determined by analyzing lists retrieved in consecutive polls) at a single GWebCache. About 80% of the GWebCaches get cache list update rates of 10 per hour or less, while a few caches receive very high update rates, upto 40 updates per hour. About 60% GWebCaches receive host list update rates of less than 1 per minute, whereas others receive update rates almost twice as much. Similarly, Figure 5(b) shows the CDF of the mean and maximum total request rates (as reported by the statistics files) at a single GWebCache. About 90%

[4] As shown by the Limewire[5] hostcount, during the period of our study.

of the GWebCaches receive an average request rate of 3000 per hour or less, whereas some caches receive extremely high loads of the order of 20000 requests per hour on an average, with a maximum of 30000 requests per hour.

This points to the disparity in the type of GWebCaches in the system. Some caches are very busy, with their lists evolving faster and receiving high request rates, whereas others are relatively lightly loaded. The servents we studied have some hardcoded GWebCaches, indicating that the request rates to these caches could be very high. This suggests poor load balancing in the GWebCache system.

3. How does the host list at a single GWebCache and at all GWebCaches evolve? What percentage of new hosts added to the GWebCaches are unique?

We retrieved the *host list* from the active GWebCaches every 5 minutes, and studied its evolution at a particular cache and in the whole system. As expected, the host list evolves much faster than the cache list in any GWebCache. During a 15-day period in our study, we saw over 300000 unique IP address:port combinations in all GWebCaches.

Figure 4(b) shows the CDF of the host update rates at all GWebCaches in the system. The rightmost line shows the CDF of the host updates received at all GWebCaches in the system. The dotted line 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 at a particular GWebCache is lower than the actual update rate at that cache. The update rate for IP address:port, unique throughout the system is much lower, almost by a factor of 10. This suggests that the same hosts (presumably ultrapeers) update the GWebCaches frequently with their IP addresses, leading to a high replication rate of the same addresses in multiple caches.

4. In the host list returned by the GWebCaches, how many hosts are alive, how many are accepting Gnutella-level connections, and how many are ultrapeers?

We sent Gnutella-level connect messages to the hosts in the host lists returned by the GWebCaches. If a TCP connection was established, we determined that the host was alive. If a Gnutella-level connection was established, we determined that the host was accepting incoming connections. Out of the hosts that responded with the proper *pong* response, we determined whether the host was an ultrapeer or not, using a field *X-Ultrapeer: True/False* in the response.

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 GWebCaches are actually accepting incoming connections. This could be a cause for the high bootstrapping times of servents in some cases, where peers waste time trying to connect to off-line hosts returned by the GWebCaches. The reliability of content served by the GWebCache system is therefore questionable.

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

5.2 Experience of a local GWebCache

We set up a GWebCache locally by modifying a publicly available PHP script for the GWebCache v0.7.5[9] to log request arrivals, and advertised it to the global caching infrastructure. Our cache received update rates of about 7 per hour to the host list and 4 per hour to the cache list, and request rates of about 15-20 per hour for the host list and 5-10 per hour for the cache list. Comparing these rates to those of other GWebCaches seen earlier, we can see that our local cache is used less frequently than the other GWebCaches.

6 Conclusions

In conclusion, 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) Although servents implement a similar structure for the bootstrapping algorithm, there is considerable variation among various implementations, that correlates to their bootstrapping performance. (2) The neighbors of a peer play an important role in the search performance of the peer. Hence it is important that the bootstrapping process results in good neighbors. (3) Even though the GWebCache system is designed to operate as a truly distributed caching system in keeping with the peer-to-peer system philosophy, it actually operates more like a centralized infrastructure function, with some GWebCaches handling a large volume of requests while others are idle. (4) The GWebCache system is subject to significant misreporting of peer and GWebCache availability due to stale data and absence of validity checks. We further aim to analyze the effect of bootstrapping 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.

References

1. Saroiu, S., Gummadi, P., Gribble, S.: A measurement study of peer-to-peer file sharing systems. In: Proceedings of Multimedia Computing and Networking. (2002)
2. Chu, J., Labonte, K., Levine, B.: Availability and locality measurements of peer-to-peer file systems. In: Proceedings of ITCom: Scalability and TrafficControl in IP Networks. (2002)
3. Ng, T.E., Chu, Y., Rao, S., Sripanidkulchai, K., Zhang, H.: Measurement-based optimization techniques for bandwidth-demanding peer-to-peer systems. In: Proceedings of IEEE Infocom. (2003)
4. Oram, A.: Peer-To-Peer: Harnessing the Power of Disruptive Technologies. 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.com)
9. : Gnutella Web Caching System. (http://www.gnucleus.com/gwebcache)
10. : Ultrapeer Specifications. (http://www.limewire.com/developer/Ultrapeers.html)
11. : Windump. (http://windump.polito.it)