Gnutella Simulator

Denis Petrussenko
Florida Institute of Technology
Department of Computer Sciences
Melbourne, Florida, USA
dpetruss@fit.edu

Abstract

Peer-to-peer network in general and specifically the Gnutella network suffer from poor search facilities due to their distributed and decentralized nature. Several improvements to the current implementations have been proposed, but the development of new improvements is hampered by the lack of a simple Gnutella simulator to test out algorithms on. The simulator that we developed is easy to use and provides detailed statistics on what is going on within the network.

1 Introduction

Searching a peer-to-peer (or P2P) network is not an easy task. Because there is no centralized server that keeps track of which nodes have what content on the network, each node has to figure out on its own what data is available to it. The original Gnutella specification [3] calls for a very simple search mechanism using broadcast messages. Unfortunately, after it gets past a certain size, these broadcasts flood the entire network and result in massive bandwidth waste. Several schemes have since been proposed and they are getting more and more efficient. Local indices [1] was the first such improvement, designed to cache the indexes of the neighboring nodes in order to reduce the number of hops a message must be sent over to find a valid result. Dynamic index allocation [2] is the latest development, creating indices only on the nodes that have quick access to popular data. However, the performance of all these algorithms depends on the network being relatively stable, with nodes dropping only rarely and even then with a notification. When introduced to the world of mobile agents, such as cell phones, the performance of all these algorithms drops dramatically. Can anything be done to improve the performance of these algorithms in an ad-hoc environment? We look at the effect of adding AntNet-style [3] routing to a Gnutella client that utilizes the Dynamic Index Allocation scheme. Our experiments show a clear improvement in the number of search results satisfied as a result of this.

This paper is divided as follows. Section 2 describes the related work. Section 3 looks at the implementation of the simulator. Section 4 presents some results obtained with the simulator. Finally, Section 5 provides some concluding remarks and future work.

2 Related Work

The original Gnutella specification [3] calls for a simple recursive search. A Query message containing a list of keywords that this search is looking for is sent out to all neighbors of the current node. Unfortunately, after it gets past a certain size, these broadcasts flood the entire network and result in massive bandwidth waste. Several schemes have since been proposed and they are getting more and more efficient. Local indices [1] was the first such improvement, designed to cache the indexes of the neighboring nodes in order to reduce the number of hops a message must be sent over to find a valid result. Dynamic index allocation [2] is the latest development, creating indices only on the nodes that have quick access to popular data. However, the performance of all these algorithms depends on the network being relatively stable, with nodes dropping only rarely and even then with a notification. When introduced to the world of mobile agents, such as cell phones, the performance of all these algorithms drops dramatically. Can anything be done to improve the performance of these algorithms in an ad-hoc environment? We look at the effect of adding AntNet-style [3] routing to a Gnutella client that utilizes the Dynamic Index Allocation scheme. Our experiments show a clear improvement in the number of search results satisfied as a result of this.

This paper is divided as follows. Section 2 describes the related work. Section 3 looks at the implementation of the simulator. Section 4 presents some results obtained with the simulator. Finally, Section 5 provides some concluding remarks and future work.
bandwidth used to transfer these messages could instead be used to transfer files. The local indices scheme [1] was an improvement over the original Gnutella specification. The idea was that instead of each node keeping an index of just its own files, it would also keep an index of all the data in nodes within r hops of itself. Instead of every node responding to a Query message and then broadcasting it to all of its neighbors, a policy is set that specifies at which hop counts nodes respond to Query messages. For example, with policy P = 1,6, only nodes that receive a Query with a hop count of 1 or 6 actually respond to it. However, if the hop count is less than the largest number in the policy, the node still forwards the Query message to its neighbors after incrementing the hop count. When a Query message is received with a hop count equal to the largest number in the policy, that message is processed and disregarded. This approach results in faster search times, because each node can process a Query based on data from all the nodes within r hops of itself, so search results come back much faster and in several big chunks, according to the policy implemented. However, the problem with this approach is that extra messages must be sent back and forth between nodes to make sure that all their indexes are up to date. Dynamic index allocation [2] was introduced in order to use bandwidth more efficiently. Instead of making an index of all data within r hops, each node looks at the Query and QueryHit messages that are passed across it and dynamically decides to create an index after enough messages are observed in a time period. It continues to monitor traffic coming through it, deciding to stop being an index after traffic values drop below acceptable levels. This scheme has an additional benefit in that it is able to index data outside of r hops of itself. This is done by observing QueryHit messages that come from outside of r hops and using the metadata from them in the index. The drawback, like with the other indexing schemes, is that extra messages are required to create and maintain indices. However, this approach is still much better than the previous ones.

3 Approach

There are many ways to evaluate the performance of any network, and Gnutella is not any different. One of the more popular ways is to use a network simulator that can approximate a network inside one computer. However, there are currently very few choices as far as network simulators go. We found that none of the current approaches were simple enough to use, without sacrificing a significant amount of time to adapt the simulator to the Gnutella specifications. A new network simulator called GnutellaSiml was written, tailored specifically for simulating traffic on the Gnutella network. To approximate the real world, several key factors had to be considered. Actual clients ship without much information about the outside world, so they need to bootstrap the first time they used. Gnutella uses the web caches approach, where there are a number of caches that run on regular web servers that keep track of Gnutella nodes. A client does ship with one or several web cache addresses, which it uses to find other nodes upon first initialization. To approximate this behavior, our simulator contains a WebCache object. There can be any number of these objects and nodes are initialized with one of them, currently chosen at random. Having more than one cache has the effect of splitting the network into islands, where nodes are only aware of other nodes that use the same cache. In the real world, caches talk to each other to share nodes information. This node information sharing is not implemented in our current approach, but can be easily added. After a node first starts up, it is ideally connected to several other nodes. The exact number of connections it tries to open is usually dependent on the amount of bandwidth available to that node. More connections usually mean more data available to a node, but a high connection count can also overwhelm a network connection with limited resources. Nodes in GnutellaSiml have upstream and downstream bandwidth and the amount of nodes they will try to connect to is a function of upstream bandwidth. After connecting to at least one other peer, the node is now able to search for and transfer files, as well as share its own data. In the real world, the user would proceed to send out one or several searches as soon as their node is operational. To approximate this behavior, each node has a list of topics that is it interested in. When a connection is established, the node proceeds to send out Query messages for these topics at some interval, usually at least every 10 clock ticks. Nodes have an internal limit on how many queries they will send out at a time, which is currently set to two. It can be changed on a per-node basis to reflect greedy users in the real world. To search for something, a node has to have a concept of
something implemented in it. In the real world, some-thing is represented with files and users search for words
that are matched against the file names on the nodes that
respond to the search. This was approximated by each
node keeping a list of files that it contains. Each file has a
type, a name and a size. Actual file contents are irrelevant
in a simulation, so they are not expressed in any way. File
name and type are expressed with one byte each. This
was done to keep the number of possible files and types
to a reasonable level. If file names were random strings
of arbitrary length, then each node could potentially have a
list of files that are unique to it. This does not truly reflect
the real world, as usually there is a set of users that have
some popular data that other users then search for. Single
bytes were used to keep the number of possibilities down,
so that there would be several nodes with the same data
that others could then look for and find. Files types were
employed to represent the fact that real users are usually
interested in a certain type of file. For example, if there is
an image and an executable with the same name, a user
will generally go for one over the other, depending on
the type of data they are looking for. File types are also
useful for modeling user interest. Currently, each node
is assigned a random list, between 5 and 15 items long,
that it is interested in. Then this is the only kind of files
that node is going to be searching for. These can be fine-
tuned to reflect the different demographics of real-world
users, but currently they are just randomly selected. After
a search query goes out, users want results to come back
as fast as possible. The original Gnutella specifications
led to insufficient search times, with results coming in af-
tinations left that the and has not yet been to. Ants that either reach the aforementioned dead end or travel the desired amount of hops flip their travel flag to backwards before they are sent back. Then on the way back they update the routing table inside each node with the node information they carry. This results in each node having a fairly accurate table of network links in its vicinity. The dynamic adaptive indexes code was then modified to use the table that AntNet builds and keeps updated to replace failed nodes with a peer they were directly connected to that has the highest upstream bandwidth. That way when a node goes offline and the peers it is connected to detect that, they can replace it with the best neighbor rather than going out to the web cache to collect more peers or relying on the ping / pong messages to discover better peers in the surrounding area.

4 Evaluation

To compare performance of different indexing approaches, there are several metrics. One of the most popular is the number of messages passed during each step in the simulation was used. This is directly related to the amount of bandwidth the system utilizes, so the lower this value, the more efficient the system is. Figure 1 represents the total number of Query messages sent out per time period for each of the indexing approaches discussed.

The node depth of 7 for the original approach is in sharp contrast with the other approaches, which all use a maximum depth of 5. The extra two hops result in a much higher number of query messages being sent at all times. The ratio of query hits to queries is another good metric. Ideally, every query would produce a response, because queries that do not get a response are essentially a total waste of resources. This is of course impossible to accomplish 100% of the time, as it would require advance knowledge of the search results. However, it is still a useful approximation of how well each approach provides relevant results. Figure 2 shows the amount of query hits received per time period for each indexing approach.

One other metric is the amount of actual results returned per time tick, across all nodes. The higher this number, the more relevant the search results are and the more users are satisfied with their search results. Figure 3 shows the number of total files found across all nodes that received a query message per time iteration.

On the same note, figure 4 shows the difference in total result counts when using the Dynamic Local Indices with AntNet and applying two different failure rates on the network. The blue squares represent result counts when the overall failure rate is low (0.6% for low bandwidth nodes, 0.01% for high bandwidth nodes) versus result counts when overall failure rate is raised to 6% for low bandwidth nodes and 1% for high bandwidth nodes.

The reason that higher failure rates lead to higher result counts is that failed nodes are replaced with higher bandwidth nodes, thanks to AntNet routing, which in turn are connected to more nodes, leading to more files being available to more nodes.
5 Conclusion

We have developed and implemented a simulator for the Gnutella network with three different indexing approaches, as well as added AntNet routing to the most efficient current approach. Results obtained from our simulation allow comparisons to be made between various search indexing approaches as well as the fault tolerance improvement that AntNet provides, among other things. Current limitations include lack of scalability past several thousand nodes and the lack of asynchronous communication. Future work will make use of the TCP/IP stack to allow the simulator to scale across as many machines as desired.

References

