PBJ: A Gnutella Inspired File Sharing System

Camden Clements
School of Computing
Clemson University
Clemson, SC 29632
Email: camdenc@gmail.com

Adam Hodges
School of Computing
Clemson University
Clemson, SC 29632
Email: hodges8@clemson.edu

Zach Welch
School of Computing
Clemson University
Clemson, SC 29632
Email: zwelch@clemson.edu

***Abstract*—PBJ is a distributed file sharing system designed on many of the same principles of the gnutella file sharing application. Rather than route search requests through some central entity like Napster and, to a lesser extent, BitTorrent, PBJ creates a network of connected users. Search requests are broadcast between users and files are downloaded directly between them. This paper details the algorithms behind PBJ, the specifics of its implementation, and a comparison of PBJ to other popular file sharing systems. Future improvements to PBJ are also discussed.**

I. Introduction

Peer to peer (P2P) file sharing applications are some of the most well known and ubiquitous distributed systems today. These applications allow users of their system to share and download files between each other (usually) free of charge. In the last fifteen years, file sharing has gone from virtually nonexistent to one of the largest sources of Internet traffic [SOURCE]. P2P system structure can vary wildly from highly centralized, in which a single server handles all requests, to highly decentralized, where there are no central elements and the user client programs work together to find results. Beyond basic topology, there are a host of design trade offs. We were also interested in creating a system that would not be considered liable if illegal content were shared on it. This has been a problem for file sharing applications in the past, particularly Napster. PBJ is a distributed file system that has been designed to be stable, reliable, and able to adapt to changing needs of its users. Due to these design goals, we decided to loosely base the design of PBJ on Gnutella, a highly distributed file sharing application. While our overall design is Gnutella based, we looked at a range of famous file sharing systems, and several of them influenced our decision making during the design process.

A. Background

1) Napster: A defining characteristic of a P2P file sharing application is the kind of network topology it employs. Napster, one of the first file sharing networks, was a highly centralized system. All online Napster users connected to a central server. Each user would have a folder designated for sharing audio files (only mp3s were transferred on Napster). This central server would handle a user request by searching for related files in the share folders of other users. The central server would then report a list of relevant files and their locations that the user could choose to download. Any files the requesting user chose to download would directly connect with the associated machine for actual file transfer. The ability to search for and download music free of charge was very popular and the Napster user base was very large. It also happened that a large percentage of the downloads on Napster constituted copyright infringement, since users were distributing music without the creators consent. From a legal standpoint, the flaw in Napsters design was that all of the files on the system were indexed on the central server, including files that broke copyright. This meant that legally, Napster knew about and facilitated copyright infringement. In 2001, when the inevitable flood of copyright infringement lawsuits were filed against Napster, a court order shut down the central server. Because it is a central server, the network ceases to exist when it is taken down. File sharing applications developed after Napsters shut down attempted to avoid legal liability by making their applications less centralized and by not having their computers keep track of individual files.

2) BitTorrent: With the dismantling of Napster, file sharing applications started to become more complex and decentralized. Most file sharing today involves the BitTorrent protocol. BitTorrent differs from other file sharing systems in that it downloads file segments from multiple sources instead of a single file transfer between two users. BitTorrent works by having a user obtain a torrent file (usually by downloading it from a website). The local BitTorrent client interprets the contents of the torrent file and connects with a tracker, a server with information about how to find the file. The tracker finds seeders, other BitTorrent clients, with a local copy of the file to be transferred. The file also identifies the swarm, a set of clients with a portion of the file, usually in the process of downloading it themselves. The searching client downloads from these sources simultaneously to build the desired file. BitTorrent is especially useful for downloading popular files, since there will be a large pool of seeders and a large swarm. While BitTorrent is clearly more distributed than Napster, since there are multiple central servers instead of a single large server for the entire application. However, if the tracker(s) in a torrent file are taken down, there is no way that torrent file can be used to obtain the actual file.

3) Gnutella: Gnutella (a portmanteau of GNU, the free software project, and Nutella, the chocolate and hazelnut spread) is a decentralized network of interconnected users running a gnutella client called a node. Search requests are

broadcast to all of the searching nodes neighbor clients (which gnutella terms peers), which in turn broadcast the request to all of their peers as well. This goes on until the file is found or the request goes a specified number of peer hops without finding a file (much like the time to live field in an IP packet). A list of the files that match the search are displayed to the user, who then chooses the file to download. The searching node connects directly to the node with the desired file for file transfer. This hop based searching algorithm does allow for the possibility that a search request may fail to return a file in the network, especially as the network gets larger and larger. Later versions of gnutellas network introduced the concept of ultra nodes and ultra peers. Each ultra node is connected to a network of regular nodes as well as a large number of other ultra nodes, each having its own network of nodes, making gnutella a network of networks. The reason for this change is that it allows many more nodes to be reached from any one node in a smaller number of hops, making searching more efficient and the system more scalable. Nodes keep track of the other nodes they have previously connected with and attempt to reconnect to the network through these nodes. When connecting for the first time, these nodes must attempt to connect a set of nodes guaranteed to be in the gnutella network. This means that despite the highly decentralized nature of gnutella, a few centralized elements remain to facilitate the bootstrapping process. Unlike Napster, which kept information about files on company machines, gnutella does not keep track of the files transferred on their system. This means that they are not liable if copyright infringement occurs on their system.

II. Methodology

PBJ takes the key qualities of gnutella and uses them only as a starting point, rather than implementing a version of the actual gnutella protocol. Lessons learned from looking at applications like Napster and BitTorrent also played a role in the design of PBJ. PBJ functions like gnutella in that it creates a network of connected user nodes. However, it also borrows an important idea from Napster (though slightly altered). PBJ has a central server running called the gateway. While Napster kept track of files stored on the system, the gateway keeps track of the nodes in the system and controls the building of the network. The gateway allows for easy handling of stabilizing the network with nodes being added and removed regularly from the system. There are three things the PBJ network is designed to handle : building the network, searching the network, and handling node disconnects.

A. Building the Network

LIke gnutella, PBJ contains a network of ultra nodes. Each of these ultra nodes has a set of nodes that form an ultra nodes sub network. Ultra nodes in PBJ are given a unique ID starting at 0 and increasing by one each time a new ultra node is added to the network. An ultra nodes ID is assigned in the gateway when it is added to the network. This unique ID is a vital element to building the network.

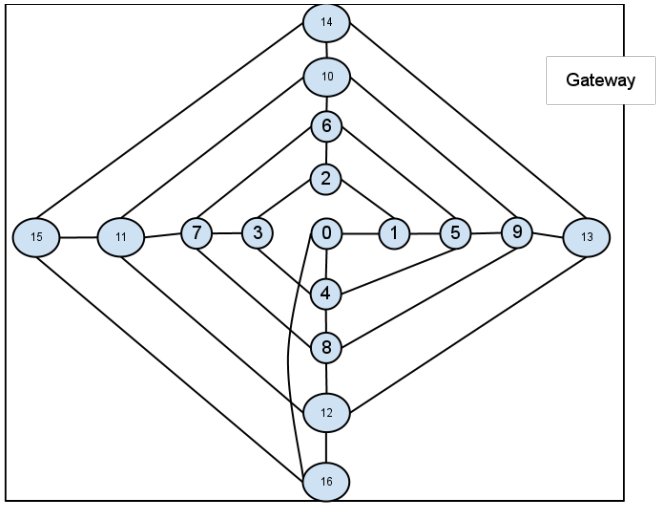

Fig. 1. PBJ Network of Ultra Peers

The basic algorithm for connecting ultra nodes in the network builds an outward spiral of ultra peers. While this alone would be wildly inefficient in terms of searching and stability, new ultra nodes also connect to ultra nodes deeper within the spiral. This allows search requests to quickly move between otherwise distant ultra nodes. More specifically, the first ultra node created will be given an ID of 0, the second an ID of 1, etc. When an ultra node is created with ID Y, it will attempt to connect to all nodes $X, 0 < X < Y$, where $\log_2(|Y - X|) = Z, Z$ being an even integer. So for example, node 16 would connect with node 15 $(16 - 15 = 1 = 2^0)$, node 12 $(16 - 12 = 4 = 2^2)$, and node 0 $(16 - 0 = 16 = 2^4)$.

A clear benefit of this structure is that the PBJ network has a high level of redundancy in the connections. This means that the nodes are connected in such a way that it is easy for search requests to reach most, if not all, of the nodes on the network in a few hops. Currently, the process of actually adding ultra peers to the network and telling them which other ultra peers to connect to is handled through the gateway. The main job of the gateway is to make decisions about how to add a new user node to the network. The gateway keeps track of the status of all the ultrapeers currently in the system. When a new node tries to join the system, the gateway makes several decisions based on the current status of the PBJ network. First, the gateway decides if the new node will become an ultra node. If so, it will give the new ultra node a list of other ultra nodes to connect to. If the new node is not an ultra node, the gateway will provide information on the appropriate ultra peer sub network to join. These decisions by the gateway essentially dictate how the network is built. We devised two separate algorithms for building the network; the one we actually implemented is described here and the other is detailed in the section Future Work. As implemented, the first node to enter the network becomes ultra node 0. If, for example, there are three nodes per ultra node, the next three nodes to connect would become node 0s sub network. The

fifth node to join the network would become ultra node 1 and would connect to ultra node 0. In this scheme of building the network, each ultra nodes sub network is filled before the next ultra node is created. This algorithm ensures that the graph has the smallest number of ultra nodes possible in a system and all but potentially the last ultra node have filled sub networks.

B. Searching the Network

The PBJ client allows a share path to be passed as a command line argument. The default share directory is a subdirectory named share. All files and sub directories in the users share directory will be available to the network for other nodes to download. When a user wants to search, they enter a keyword and submit a search request. The search request will be sent to the searching nodes ultra node (unless the node is itself an ultra node).

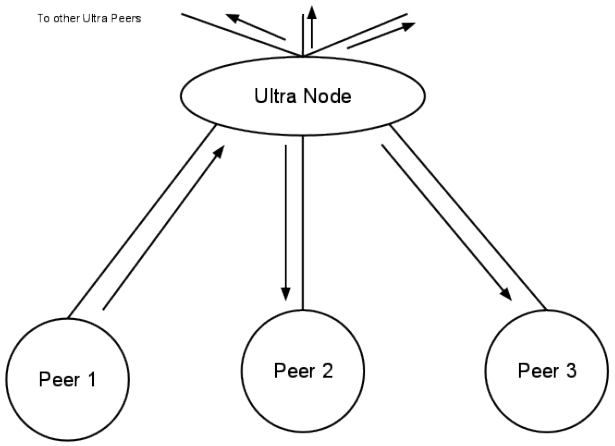

Fig. 2. Searching For a File

The ultra node will send the search request to all of the other peers in its sub network before broadcasting it to its connected ultra peers. Search requests consist of four major parts - the searching nodes address, the keyword, a search id, and the time to live (ttl). The ttl is initially set to a positive integer. Each time a search request is passed between ultra peers, the request ttl is decremented. Notice that ttl only limits the number of hops among ultra peers, passing a request within an ultra node and its sub network has no effect. If an ultra node receives a search request with a ttl of 0, it will send the request to its sub network and delete the request. If the keyword is found in any part of a filename, the finding node sends a message to the search node with its IP address and the path of the desired file relative to the share folder. As these acknowledgement messages come into the search node, it displays to the user a list of all the files found on the network. The user can then choose one or more of the files to download. The searching node directly connects to the node with the selected file and downloads it.

The search ID was added as part of the search request as an attempt to significantly lessen the search request traffic on the network. The structure of the PBJ network means there are many different paths between any two ultra nodes. While this is very useful in keeping the network stable and allowing search requests to quickly reach a large number of files, it also means that the same ultra node will receive and process a request multiple times, sending the message out to all of its ultra peers, who have already received the request. This type of network behavior is extremely inefficient. PBJ attempts to fix this problem in part by having a unique pair of values in each search request. One of these values is the IP address of the requester and the other is a search ID that is local to each node and represents the number of requests previously sent out by that node. Together, this (address, search ID) pair make a unique request identifier. Each node keeps track of the last several request ID pairs received and if the request ID pair of a newly received search request matches one in the list, that request is ignored. This scheme vastly cuts down the amount of redundant request traffic and makes the system more efficient as a whole.

C. Disconnecting from the Network

An area of contention during development of PBJ was how to correctly handle ultra peer disconnection. Various solutions were proposed, including updating the gateway, electing a node from the sub network to be a new ultra node, and the Ostrich approach (ignoring it). We eventually decided to implement a system of having the gateway check on existing ultra nodes. Upon receiving a request to add a new node to the network, the gateway pings every ultra node in the network. If one or more does not respond, the gateway marks those as missing, and inserts ultra nodes into these positions until they are all filled. The point here is to ensure that important connections in the network are rebuilt before attempting to refill the sub networks of ultra nodes. In addition, ultra nodes will occasionally ping their ultra peers and sub network. If an ultra node does not get a response from an ultra peer, it removes it from the list of connected ultra peers. If an ultra peer cannot connect to a peer in its sub network, it removes that peer from its list and notifies the gateway that a spot has opened up in its sub network. If a node in a sub network cannot reach its ultra peer, it re enters the network through the gateway. This ensures that all lost nodes will be replaced as new ones are added to the system, and that the network will not become fractured with high node turnover.

III. Results

All of the PBJ network communication is facilitated through the use of the Flask python module. Flask is a web microframework that is designed to simplify the development of Python web applications. HTTP was chosen as a protocol over TCP due to its reliability, simplicity, and flexibility. Flask trivializes the implementation of custom HTTP responses. Every node, as well as the gateway, is running an instance of a Flask server. The server is configured to run on port 5000 as a default, but this could be easily changed. Flask was not available as an installed Python module on the lab machines that we were planning on using for our testing, and

we did not have administrative privileges to install it, so we needed a way in which we could install and use Flask on every lab machine without modifying any system files. The solution was to use virtualenv, which is a tool used to create virtual Python environments. A virtual Python environment can be created in a users home folder on a system using virtualenv, and then that user has permission to modify that environment however they wish. In this way, we were able to deploy a Python environment with the Flask module and its dependencies installed on every lab machine.

A concern of ours when laying out the specification of how nodes would communicate was the case in which a node would have to communicate across a firewall. Firewalls add a great deal of complexity when designing a distributed system that is intended to be accessed by a large number of computers under diverse network constraints. Our ultimate decision was that since our project is academic in nature and limited in resources, we were going to ignore the cases in which communication over port 5000 is unavailable to a node.

A. Test Plan

In order to claim that we have designed a stable and scalable system with confidence, we needed to design a solid testing plan. In the design phase of our project, we thoroughly discussed the use of Condor, a HTC application that can deploy processes to a large number of computers, to test our network stability and search algorithms for large numbers of nodes. Unfortunately, the computers on Clemsons Condor network are equipped with Windows Firewall, which blocks the access of ports that have not been explicitly granted permission. As previously discussed, we will be ignoring the cases in which our nodes are behind firewalls, so this test strategy was inadequate. Instead, we devised a test strategy in which we wrote a script to connect every available lab machine we had to our network.

The end result was a test environment in which over 70 search-able nodes with unique files were connected to our network. Although this is a considerable amount of nodes, we found that it was not sufficient to fully test the scaling and performance of our search algorithm. We used this large network of connected nodes primarily to test the functionality and reliability of our system. The main goals of testing the gateway server was twofold. First, correct network topology was to be maintained when new nodes were added. This could easily be checked by feeding the gateway a large number of pseudo nodes to verify that the intended topology could be maintained for any size network. Furthermore, an applet was developed to display an image representing the network graph, making it easier to see node to node connections. The second goal of gateway testing was to verify that any dropped node notifications were being handled correctly, which involved updating the network structure for addition of future nodes. This is further discussed later when testing node functions.

Testing node functionality proved to be much more in depth. For this, we developed a series of repeatable test cases that the network should be able to handle. These test cases included: initiating a search as an ultra node, initiating a search as a node, searching from one end of the network to the other, searching from the middle of the network to either end, and downloading a search result. We also monitored the gateway to ensure that the network structure was maintained and rebuilt after all node disconnection scenarios: dropping a node from the beginning, middle, and end of the network, as well as dropping an ultra node from the beginning, middle, and end of the network. We repeated these test cases after every change we made to ensure that PBJ was stable and working properly.

IV. Analysis

Our testing demonstrated to us the stability of the network. To a large extent, the main goals of the project were met. A network of connected peers could be created using only a simple centralized gateway, while keeping file searching and downloading independent of such a server. We discovered, however, that there was a large amount of traffic on the network between the pinging of peers and redundant searching, even reduced as it is with the unique request ID. We also feel that an explicit ttl will eventually limit PBJs scalability. This section provides an in depth look at our test results and possible improvements future versions of our system could implement to deal with these perceived issues.

A. Future Work

1) Decentralization: One improvement previously discussed is changing how nodes join the system. The current gateway method functions well, but its centralized nature is hardly ideal for a system whose design so highlights decentralization. Several alternatives have been discussed. The most likely implementation would be to place the burden of network building on the existing ultra nodes. A node would keep track of its previously connected ultra nodes and attempt to connect to these peers. Depending on its current status, the ultra peer might add the new users or decide to push it towards higher valued ultra peers to attempt placement. Eventually it would either get placed in a sub network or reach the ultra peer with the highest id and become the new highest ultra peer. If none of the previously connected nodes are currently connected to the network, a range of options could be implemented to ensure the user gets into the network including but not limited to: keeping the gateway as a backup, having ultra nodes at known URLs, placing the burden on the user to find the information.

2) Network topology: Several different specific algorithms for how exactly the network is organized were discussed. The current algorithm, described above, creates a new ultra node and then fills that ultra nodes sub network before creating a new ultra peer. While this algorithm works acceptably, it ignores several observations about PBJs network. In PBJ, the number of hops it takes to get from node A to node B is highly dependent on the current number of ultra peers in the system. As a simple example, the optimal number of hops from ultra peers 0 to 63 is 5 hops (0-16-32-48-47-63) if ultra node 64 does not exist and 2 hops (0 -64-63) if ultra node 64 does exist. Indeed, a node whose largest hop was one of the

last nodes to be added has the optimal network structure for searching.

Our algorithm for taking advantage of this trait slightly alters how the network is built by the gateway. The basic idea behind this algorithm is to have an ultra peer back bone in place first with empty sub networks and then fill in the sub networks. When creating a new network, the first eight (eight is half of sixteen) nodes into the system will become the first eight ultra peers. The next nodes to join the system will join these ultra peers sub networks. When all eight sub networks have been filled, the gateway then add the next nodes as ultra peers in the system until there are thirty-two ultra peers (thirty two is half of sixty four) in the system. This continues so that when the network is full, it grows to half the size of the next even power of two before filling in the ultra peers. This ensures that the majority of the peers are close together but have the ability to move farther because there is an ultra peer structure in place to facilitate such movement. Another potential improvement would be to choose ultra peers based on network quality. Ultra peers have to handle the vast majority of the network traffic. It then makes sense that the machines with the strongest network connection should be ultra peers. This would help stabilize the network and improve quality.

3) Searching: We also feel the search algorithm could be improved or a new algorithm implemented. Going forward, it seems that the ttl is the main issue to be addressed. Its purpose is to limit system traffic, but if the network becomes large enough, it is possible searches from a certain node will never reach other machines far enough away in the network. Though additional testing would be necessary, we would need to see the effects of larger ttl values on system traffic. It may make sense to remove the ttl all together and count on the unique ID checking to limit redundant network traffic. We have also discussed tiers of ttl values corresponding to the size of the hop to be taken. This algorithm is still early in development and would need more research.

4) Stabilization: Handling node failure was a tough challenge in early development. As convenient and stabilizing as our node ping method for detecting disconnected nodes is to the network, it leads to a lot of network traffic. Currently, nodes ping each other every few seconds. It might make sense to have nodes ping only right before sending a search request, or to implement some kind of TCP style application level three way hand shake. In addition, it might make sense for ultra nodes to contact the gateway when they have lost contact with an ultra peer, rather than having the gateway ping all ultra nodes in the network.

5) User experience: Simple things such as an improved GUI and security/privacy features could also be added to make the system more user friendly. Another concern we have is attempting to mitigate is leeching, the practice where by users of a file sharing network do not share any files to be downloaded but still use the network to get files.

V. Conclusion

PBJ is a decentralized file sharing system whose highly connected nodes, search algorithm, and intelligent node loss handling make the ground work for a stable and reliable application. PBJ is heavily influenced by existing file sharing services gnutella and to a much lesser extent Napster. PBJ users connect to the system through a dedicated gateway, either becoming ultra nodes connected to other ultra nodes or part of an ultra nodes sub network. Searching the network is done by keyword comparison to filenames, and search requests are broadcast between ultra peers.

Results are displayed to the user to choose a result to download. The gateway and the nodes themselves periodically ping their connections. If an ultra peer does not respond to a ping, it is removed from the ultra nodes active connections. If a sub network nodes ultra node does not respond, the node will exit the network and attempt to reconnect through the gateway. Every time the gateway attempts add a node, it first checks that existing ultra peers are connected. If one has gone down, the node to be added is added in that ultra peers place. PBJ uses Flask for network communication.

Thorough unit testing of both gateway server and node application functions proved the functionality of the PBJ network. It also let us see what areas of the sharing system need the most attention in order to improve performance. Network scalability and stability were seen to be correct and fully functional as intended. Based on this testing, however, we have a number of future improvements to PBJ. These improvements would increase the networks stability, decrease redundant traffic, and reach a larger percentage of the network.

Acknowledgment

The authors would like to thank Dr. Goasguen for his help and advice, the Clemson University Systems staff for quickly responding to the various technical problems and hurdles faced in setting up machines for testing, and Dunkin Donuts for making their delicious coffee the authors used in the majority of this project.

References

[1] H. Kopka and P. W. Daly, *A Guide to LaTeX*, 3rd ed. Harlow, England: Addison-Wesley, 1999.