http://ansinet.com/itj

ITJ

ISSN 1812-5638

INFORMATION TECHNOLOGY JOURNAL

ANSInet
Asian Network for Scientific Information
308 Lasani Town, Sargodha Road, Faisalabad - Pakistan

Information Technology Journal 5 (1): 1-6, 2006
ISSN 1812-5638

The Shadow Protocol: A More Optimized Version of Gnutella

Muhammad Rashid and Khalid Rashid
Department of Computer Science, Faculty of Applied Sciences,
International Islamic University, Islamabad, Pakistan

Abstract: This study presents an analysis of the Shadow protocol, a type of peer-to-peer networking model that currently provides decentralized file-sharing capabilities to its users. The shadow protocol has been designed to be a more optimized version of the Gnutella protocol. Initially, the study explains the basics of the peer-to-peer networking and then compares the two types of this networking standard: centralized and decentralized. The shadow protocol is classified as a decentralized model. The Napster and Gnutella protocols are briefly described as well. A detailed description of the Shadow protocol is presented. And a discussion is provided which proves the Shadow protocol to be a more optimized version of the Gnutella protocol.

Key words: Shadow protocol, Gnutella protocol, peer-to-peer networking

INTRODUCTION

In early 1999, Shawn Fanning, an undergraduate student at the Northeastern University in the United States, started a phenomenon called Napster[1]. Fanning envisioned Napster as a service that allows users of his system to list the MP3-encoded music files that they are willing to share and let other users download them through the Napster network. The central computer would then at all times have an up-to-date master list of files that people are willing to share and the list would be updated by the users' software as they log on and off the system.

Fanning's idea required a network infrastructure of servers for the centralized peer-to-peer data access and data storage and a corresponding bandwidth for allowing a large number of user connections. After his idea is implemented, Napster at one point attracted over 30 million users with over 800 thousand of them accessing the network simultaneously[1]. The upper limit on the number of users of Napster is only imposed by the bandwidth, as the service itself never lacks popularity.

Unfortunately, due to violations of the copyright laws and by the order of the Supreme Court of the United States, Napster is forced to impose complex limitations on the shared files thereby downgrading the quality of their service. Finally, in July of 2001, Napster is forced to shutdown its servers due to software-related problems that occurred as a result of Napster's developers trying to ensure that these limitations are imposed throughout their network.

Beside the copyright infringements, the reason why the Napster network fails to provide Quality of Service (QoS) to its users is due to its centralized peer-to-peer character. Since Napster has a single point of entry, the network can completely collapse if its central point becomes incapacitated. In addition, this central point has a complete authority over the data distributed through the network and is solely responsible for its contents (Napster's legal problems stem from this fact). A solution to providing QoS in the peer-to-peer environment is by using a decentralized model instead and having multiple access points, some of which if incapacitated would not incapacitate the entire network. One such decentralized peer-to-peer model is the Gnutella network based on the Gnutella protocol[2].

Gnutella is a decentralized P2P file-sharing model developed in the early 2000 by Justin Frankel's Nullsoft (AOL subsidiary and the company that created the WinAMP MP3 player)[3]. Gnutella's development was halted shortly after its results were made public and the actual protocol was reverse engineered using the code that was downloaded from the Nullsoft's web site just before its closure. Today there are numerous applications (referred to as Gnutella clients) that employ the Gnutella protocol in their own individual way and that allow their users to access the Gnutella network.

To share files on the Gnutella network, a user (node A for example) starts with a networked computer that runs one of the Gnutella clients. Since this node will work both as a server and a client, it is generally referred to as a (Gnutella) servant (both a SERVer and a cliENT). Node A will then connect to another Gnutella-enabled networked computer (node B for example) and then A will announce existence to B. Node B will in turn announce to all its neighboring nodes (nodes C, D, E and F for example) that A is alive. This pattern will continue

Corresponding Author: Muhammad Rashid, Department of Computer Science, Faculty of Applied Sciences, International Islamic University, Islamabad, Pakistan

recursively with each new level of nodes announcing to its neighbors that node A is alive[3]. Once the node A has announced its existence to the rest of the network, the user at this node can now query the contents of the data shared across the network. Figure 1 further clarifies this model.

This announcement broadcasting will end when the Time-To-Live (TTL) packet information expires; that is, at each level the TTL counter will be decreased by one from some initial value until it reaches zero at which point its broadcasting will stop. To prevent users from setting this initial TTL value too high, the majority of the Gnutella servants will refuse packets with excessively high TTL value. However, from the user's perspective, maximizing the chances of finding the required file means using as high as possible TTL value therefore creating a trade-off point for this network. Where low TTL means minimizing the usage of the network resources, high TTL value means maximizing the QoS provided to the users of the network.

The optimal TTL value would then (among others) depend on the network topology and traffic characteristics for a particular location and a particular time of the day, respectively, when the query is done.

PEER-TO-PEER NETWORKING

Before analyzing the details of the Shadow protocol, the reasons why peer-to-peer networking is important and why people are considering peer-to-peer file sharing for anything else but sharing of copyright-infringing music and video files must be analyzed.

As mentioned before, Napster is a type of the peer-to-peer networking model in which each party has the same communicational capabilities and either party can initiate a communication session. On the Internet, peer-to-peer (P2P) is a type of networking that allows users with the same networking program to connect with each other and directly access each other's files.

Business advantages of using peer-to-peer networking are still being discovered[4] and the key returns are harnessed through

- Distributed processing, that is allowing users of the network to schedule batch-jobs that are processed by the computers on the network during their idle time thereby decreasing the need for new computing resources.
- File sharing, that is allowing users to exchange data directly without storing files on a centralized server thereby avoiding the need to establish a centralized server and allowing two businesses to communicate with each other directly.

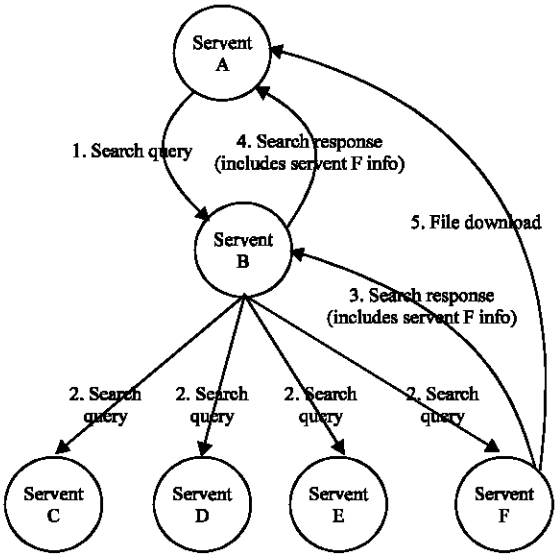

Fig. 1: Gnutella decentralized P2P model

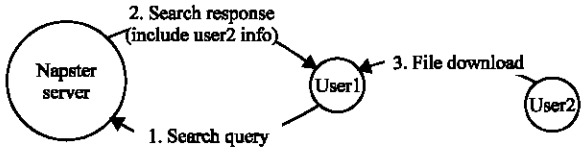

Fig. 2: Centralized P2P model

The future potential for P2P lies in its ability to change the structure of the Internet[1]. By allowing users that are accessing data from a particular web site to retrieve the cached copy of the data from a geographically closest point to their location would mean a change from the web-site centric to the purely distributed data distribution model for the Internet.

When it comes to P2P file sharing specifically, there are two models: centralized and decentralized (Fig. 1 and 2). The centralized model, used by Napster, uses a central system that directs traffic between individual users. The central servers maintain directories of the files shared by the users of the system. The servers allow query of their database and provide results that allow a user doing a query to establish a direct connection (typically using HTTP) with a user who is sharing a desired file. Figure 2 further clarifies this model.

The key advantages of the centralized model are as follows[5]:

- The existence of a central index that allows users to search and find the desired files efficiently.
- The mandatory registration of all users of the system thereby ensuring that each file query reaches all users currently connected to the network.

The key disadvantages of the centralized model are as follows[5]:

- The single entry point that creates a possibility of the entire network becoming incapacitated if this main entry point is disabled (e.g., Napster's failure under legal pressures).
- The invalid data in the database resulting from the periodical, not real time, refreshes of the data in the database.

The decentralized model respectively holds advantages and disadvantages that are contrary to the disadvantages and advantages of the centralized model. For example, having multiple points of entry in the distributed model prevents the scenario in which the entire network is disabled due to one or more of its entry points becoming incapacitated.

Another example is the lack of registration in the decentralized network thereby decreasing the breadth of data being queried and not providing the users with the same QoS as in the centralized model (Fig. 2).

The decentralized P2P (Fig. 1) file-sharing model is employed by the Gnutella protocol, so more details on this model are presented as part of the Gnutella protocol analysis.

SHADOW PROTOCOL SPECIFICATION

The Shadow protocol was developed by me (Muhammad Rashid) under the supervision of Prof. Dr. Khalid Rashid as a part of my thesis for my MS (Computer Science) degree. The Shadow protocol is an application layer protocol that passes one of several message types between peers on the network. Although it is envisioned that the protocol will be run on top of TCP and IP, this is not part of the specification. The protocol is to be used in a pure peer-to-peer environment and has been designed to be a more optimized version of the Gnutella protocol[1]. The message itself consists of two parts. The first part is a fixed length header used mainly for the routing of the message. The second part is message body, a variable length field depending on the type of message being passed. The message header is shown in Fig. 3.

The first field in the header is the IP address of the node that originally sends the message. The address is stored in IPv4 dotted format, with each value between the dots being stored in its own byte.

The UID field is a field when coupled with the IP address uniquely identifies the message on the network. This field is an 8 bit unsigned integer. It is up to the client

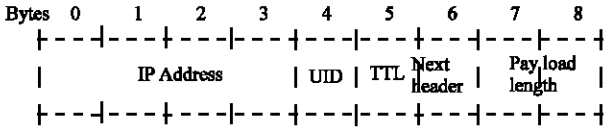

Fig. 3: The main header of the resource sharing protocol

to specify how the UID is chosen. It may seem that 256 possibilities for the UID would make unique identification impossible if a node is used for a long period of time, with 256 values being used within a matter of minutes. This is not the case however because the clients only remember the values for a certain length of time before they timeout. Because of this it is recommended that the clients repeat the sequence of UIDs once they run out.

The third field is the TTL or time to live field. This field is actually split into two sections. The first four bits are an unsigned integer value for the original value of the time to live. These four bits are never changed throughout the life of the message. The last four bits are an unsigned integer representing the actual time to live of the message. These four bits are decremented at every node. This means that for a time to live of 4, the TTL will look as follows:

0100 0100	time to live = 4
0100 0011	time to live = 3
0100 0010	time to live = 2
0100 0001	time to live = 1
0100 0000	time to live = 4

The reason for this split value for the time to live is to save space in the header whilst ensuring that any response being sent to a message will only have a horizon up to the sending node. Suppose the second node in the above example has a response to send, it will subtract the second 4 bits of the TTL field from the first four bits to set the response TTL to an appropriate value. This is shown below:

Original message = 0100 0011 time to live = 3

```
  0100
 -0011
------
  0001
```

Reply = 0001 0001 time to live = 1

It may seem that for a file sharing application a horizon of 16 is rather low, considering many nodes may only have a small number of peers. This is a valid argument when looking at a file sharing application, but this protocol is designed to be used for many different types of resource sharing. The reason why this makes a difference is due to the way people will share resources.

Table 1: Showing how the number of nodes increases with horizon (each node having 3 peers)

Horizon distance	No. of nodes
0	0
1	3
2	9
3	21
4	45
5	93
6	189
7	381
8	765
9	1533
10	3069
11	6141
12	12285
13	24573
14	49149
15	98301
16	196605

If a company is sharing processor cycles, it will most likely only want to share them within the company. This can be done by limiting the peers that are connected. All that then needs to be done is connect all the nodes within a company. Suppose each node connects to three other nodes and these nodes connected to a further three nodes. We can see how the network size increases with horizon distance in Table 1.

We can see, the problem of having a horizon of 16 can be solved using sensible peer selection. The next field is the nextHeader field. This is an unsigned integer that represents the message type that follows this header. The final field is the payload length field. This is 16 bit unsigned integer that represents the number of bytes that follow this header. This means that it is possible to have a message up to 65545 bytes long (2^{16} payload bytes + 9 header bytes).

QUERY MESSAGE

This message has the nextHeader value of 1 (00000001) in the main header. Its format is shown in Fig. 4.

The query message has a very simple format. The first byte is an unsigned integer specifying the query type. Currently the only query specified is query type 1 (00000001), the file query. The file query's query field is an ASCII string containing the elements of the file name to search for.

It is envisioned that a queries for other resources (such as spare CPU cycles) will use this message with a separate query type. Having the queries specified in such a way means that we can have up to 256 different query types, satisfying the extendibility requirement.

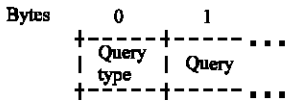

Fig. 4: The query message

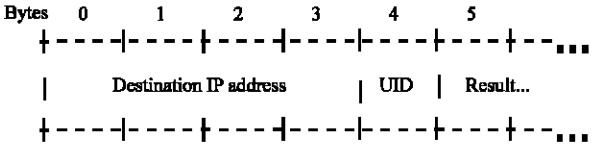

Fig. 5: The query hit message

QUERY HIT MESSAGE

The query hit message is sent in response to the query message. It has the next header value of 2 (00000010) in the main header. Its format is shown in Fig. 5.

The first two fields of the query hit message are the same as the first two fields of the main header that contained the original query. They are included so that nodes can check to see if the message is destined for them. The next field is the result field. The format of this field is dependent on the original query that was made. Again, the only specified result field is for file queries. Its format is as follows:

1st Filename
1st File size (in bytes)
IP address of file owner
2nd Filename
2nd File size (in bytes)
IP address of file owner
|
nth Filename
nth File size (in bytes)
IP address of file owner

Each of these fields is ASCII strings, separated by the new line (\n) character. As you can see the IP address of the owner is repeated for every file result. This at first appears to be a waste of bandwidth, but there is a reason for it. Suppose a remote node caches every query hit message it sees. It will soon have a long list of file names/ sizes and the IP addresses of the machines they reside on. If someone then queries this machine it can search the list of cached responses to return results faster, or possibly return results that due to horizon size would not have been returned.

When looking at the query hit message it is a valid question to ask why there is not a query hit type field similar to the query type field. The reason for this is to keep the message short and simple. The client should remember all queries it sends using the IP address/ UID combination and from that remember what type of query was issued.

ADMIN MESSAGE

This message has the nextHeader value of 0 (00000000) in the main header. Its format is similar to the query message (Fig. 4) with a one byte unsigned integer field to specify the type of network admin message follows. Two such messages have been specified. The first is message type 1 (0000001) the peer query message.

The peer query message does not contain a body. It is simple the main header followed by the admin type field. When a node sees this field it knows that the query originator is requesting a list of all peers that are connected to this node. To reply there are two methods. First (if supported), the node can make a separate connection to the query originator and send an ASCII string of the nodes IP address followed by a space separated list of its peers.

The second method is to send a response along the peer-to-peer network in a similar fashion to a query hit message. The entire admin message has the format shown in Fig. 6.

The admin message type for a peer query response is 256 (11111111). The destination IP address is the address of the node that sent the peer query. Finally, the results are sent as an ASCII string in the format mentioned above.

Out of the two response types, the first is favored more because it keeps the load off the peer-to-peer network. The inclusion of the peer query response along the peer-to-peer network is to respond to users behind a firewall, or those who are limited by the number of connections that can be made.

Now we have the message formats we can discuss how messages are passed. When a node receives a message it checks to see if it has ever seen the message before. It does this by storing the IP address and UID fields of every message processed and checking all subsequent messages against this list. This list needs to be dynamic, with new message ID's being added and old ones timing out after a few seconds. If the message has been seen before it means that there is a loop in the network. Rather than process the message again it is just dropped. If the message hasn't been seen before, a copy is made for the node to process, while the TTL of the

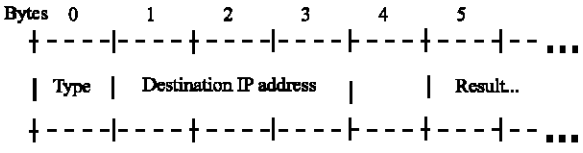

Fig. 6: The admin message for a peer query response

original is decremented. If the TTL is greater than zero, the message is forwarded. If not, the message is dropped.

Forwarding a message depends on the message type. If the message is a query, each node floods the message along all connections to all peers except the incoming connection. If the message is a query hit, the client can either flood the response along all network connections, or the more preferred method is for the client to forward the query hit along the connection that the original query came down.

ADVANTAGES OF SHADOW PROTOCOL

In February of 2001 Jordan Ritter published a paper by the name of Why Gnutella Can't Scale. No, Really[6]. This is a mathematical research paper in which Jordan Ritter has determined the amount of bandwidth that is generated by a Gnutella network. Suppose the Gnutella network is organized in such a way that it is well balanced and each node is connect to 8 other nodes and the query message has a TTL of 8 for all nodes then the amount of traffic that will be generated for one node to issue a query of 18 bytes is approximately 6.3 GB. Now if the node is performing 10 queries per second which is the average query per second during rush hour then the Gnutella network will need to transfer data at a rate of 507.2 GBps (63.4 GBps).

Since the Shadow protocol is an optimized version of the Gnutella protocol Jordan Ritter formula for calculating bandwidth can also be applied to the Shadow protocol with slight modification. Now if the Shadow network is arranged in the same way as the Gnutella network described previously and one node issues a query of 18 bytes then approximately 5.2 GB traffic is generated. If the node performs 10 queries per second then the bandwidth required will be 420 GBps (52.5 GBps). Which means that the Shadow network incurred approximately 89.6 GBps (11.2 GBps) less overhead then Gnutella performing the same query under same circumstances.

The structure of the Shadow network is decentralized, which has the advantage that there is no one point of failure. This makes the Shadow network very stable. Also when performing a search in the Shadow network it is not possible to determine exactly which node initiated the query. Therefore the shadow network is anonymous. The

Shadow protocol currently supports file sharing but has the capability to be extended to other forms of peer-to-peer computing.

CONCLUSIONS

There is doubt that the Shadow protocol is a more optimized version of the Gnutella protocol. The shadow protocol has almost all the features of the Gnutella protocol including some new features, which makes the Shadow protocol much better then the Gnutella protocol. But more work still need to be done. For example time can be spent specifying further message types to allow for the sharing of such resources as disk space and CPU cycles. Also, messages could be developed to request the 'pushing' of files from behind firewall in the same way that the Napster and Gnutella protocols feature.

REFERENCES

1. http://judiciary.senate.gov/testimony.cfm?id=199&wit_id=273 [Last Accessed 1 March 2005]
2. http://www9.limewire.com/developer/gnutella_protocol_0.4.pdf [Last Accessed 1 March 2005]
3. http://www.zdnet.com/zdnn/stories/news/0,4586,2766234,00.html [Last Accessed 1 March 2005]
4. http://www.zdnet.com/zdnn/stories/news/0,4586,2704598,00.html [Last Accessed 1 March 2005]
5. http://www.limewire.com/english/content/p2p.shtml [Last Accessed 1 March 2005]
6. http://www.darkridge.com/~jpr5/doc/gnutella.html [Last Accessed 1 March 2005]