Improving the Gnutella protocol against poisoning

Meelis Roos, Jan Willemson, Peeter Laud*
Cybernetica, Estonia
{mroos,jan,peeter}@cyber.ee

September 25, 2003

Abstract

Gnutella is a decentralized peer-to-peer file-sharing network on the Internet. One of many problems with this network is poisoning — flooding the network with false data. One kind of poisoning is serving junk instead of a piece of a file — when the downloader puts all the pieces together to get the file, the checksum does not match. We propose an addition to the protocol to protect against this kind of poisoning. We use cryptographic hash functions to prove that a certain piece of file really appears in the whole file in the specific position. The downloader can verify this against a previously distributed file checksum and discard exactly these pieces that fail the validation.

Keywords: Gnutella, file sharing, cryptographic checksums

1 Introduction

Like it or not, peer-to-peer file-sharing is becoming an important part in Internet. There are several file-sharing networks around and the amount of available peer-to-peer networking software is rapidly growing. The networks meet different problems as they grow. The goal of this paper is to solve one of these problems. The social background and legal aspects of file-sharing and peer-to-peer networking are beyond the scope of this paper.

*Supported by Estonian Science Foundation grant #5568

The first file-sharing networks were centralized (with Napster being the most famous system of centralized era). The weakness of centralized network showed itself with time and current file-sharing networks are mostly decentralized. KaZaA, eDonkey and Gnutella are perhaps the most widely known decentralized networks to date but there are also many others. Mojo Nation and BitTorrent are a couple of non-traditional and border-stretching technologies in peer-to-peer networking. It is also worth mentioning that there are usually several different software solutions to access one common file-sharing network. We are looking more closely at Gnutella network [Cli01] since this is a fully open network — the protocols are public and most clients are also open-source.

2 State of the art

There are several services that a file-sharing network usually offers: searching for files, storing of files, transport of files and proxying of connections. A file-sharing network does not need all of them for sure, e.g. BitTorrent [Co03] has no search functionality. We concentrate on file transport service in this paper since it usually takes the most of network bandwidth.

There are several techniques to save network bandwidth and use it efficiently. The most obvious method is to compress the data. This, unfortunately, helps only for the control channels but not file transfer channels since the files transported are usually already heavily packed. Another method is to recognize several copies of a file on the network and use this knowledge to make better downloading choices. We can use the fastest server to download the file or to get part of the file from one server and continue from another server when the connection breaks. We can even download several pieces of the same file from different servers at the same time to efficiently utilize our network bandwidth and save time.

For this approach to work, we need to identify the files somehow in order to find the duplicates. The simplest method — using file name and length — works but is not perfect because the users like to use different naming conventions etc. So a better mechanism is needed and the most commonly used solution is provided by checksums. The Gnutella network uses SHA-1 cryptographic hash function [SHS95] as the checksum to uniquely identify files. With each search result, not only the name but also SHA-1 hash is returned. The searches also accept SHA-1 hashes to help finding the duplicates.

3 Problems

There are many open problems in current decentralized file-sharing networks, scalability being one of the most important ones. The searches are performed as broadcasts and this limits the effective size of the networks before getting too slow. Nodes with low network bandwidth are also a problem as they become the points of congestion when faster nodes transfer their information through them. This is currently being solved by changing the structure of the network to put slow nodes as leaves to faster nodes (the latter ones being called ultrapeers).

There is also a persistent problem about the balance between content providers and content consumers. The network becomes inefficient when the percentage of sharing nodes becomes too low. Different networks use different methods to fight this problem, for example the automatic sharing of downloaded or currently downloadable content (BitTorrent), micro-payments in e-cash (Mojo Nation), better connectivity for users who share something (eDonkey [eDon]). Gnutella does not currently force the user to share anything and this might hinder it less efficient.

Another recent problem is poisoning which essentially means flooding the networks with false data. Poisoning may happen because of bugs in file-sharing software but it is usually deliberate. This problem is not a natural phenomenon of the networks, but usually an artificially created one by companies who dislike file-sharing networks. There are different ways of poisoning the network. One way is to serve files full of trash but with names that people want to download (known as file name poisoning). Another way is to pretend to serve the right file but actually send fragments of bad data so the client gets a broken file when it reassembles all the pieces that it has downloaded from different servers. This is called content poisoning and this is the problem we are trying to fight. Chen and Schroeder [CS02] have concluded that poisoning, when combined with other methods, may be quite effective in shutting down peer-to-peer networks, because users grow frustrated and leave.

In order to estimate the quality of our solution, we must first determine the extent of poisoning. If the solution would require more additional bandwidth in the Gnutella network than the amount currently lost because of poisoned data, it would not be practical. We have not done extensive experiments to find out this percentage (and it would be rather complicated to do so), but our subjective experience suggests that about 1% of the traffic in Gnutella is poisoned. Hence we will use this threshold in our future estimates.

4 Solutions to content poisoning

The first solution that already somewhat works is to overlap the different download chunks a little (like 256 bytes). The downloader can then check that the overlapping data matches already downloaded data. This usually works against buggy client programs and false files in antique client programs that do not support SHA-1 hashes for file identification. Still, overlapping is not efficient against deliberate poisoning since the poisoner can send correct overlapping bytes and still send random data in the middle of the chunk.

Hence we need to authenticate the pieces of a file to actually appear in the whole file in the right offset. The main idea of our solution to this problem is to compute a more structured checksum of the file and distribute it along with search results like SHA-1 and provide a proof along with each downloaded chunk that the hash depends on the chunk located in the file at this offset.

It must be hard to fake the proof so that a proof may realistically be given only for correct chunks. The proof must also be small enough as not to cause too much extra network traffic. In essence, we must make the proof size smaller than the current amount of network bandwidth wasted because of poisoning.

The problem we are trying to solve can be seen as an instance of *secure publication* [DGMS01]. Here the clients make queries against a large dataset, the queries are answered by publication authorities who themselves are untrusted, but are in the posession of the whole dataset compiled by a trusted authority. The clients want to have confidence that the results of their queries really come from that dataset. The trusted authority has published a checksum of the dataset, so whenever a publication authority sends out a reply, it has to complement it with a proof that this reply is a result of running the given query against a dataset with given checksum.

In our case, the downloader places its trust onto the filename, together with the checksum of the file (i.e. it assumes that name poisoning has not occurred). The file represents the entire dataset and pieces of the file are replies to the requests for specific fragments of the file. The downloader wants to be sure that each downloaded fragment really comes from a file with a certain checksum. So, in our solution we are basically applying known tools (from the area of secure publication) in a novel context.

5 Authentication trees

Authentication trees were proposed by Merkle [Mer80] and can be used to provide short evidence that some piece of data belongs to a large data collection. An example of an authentication tree is depicted in Figure 1.

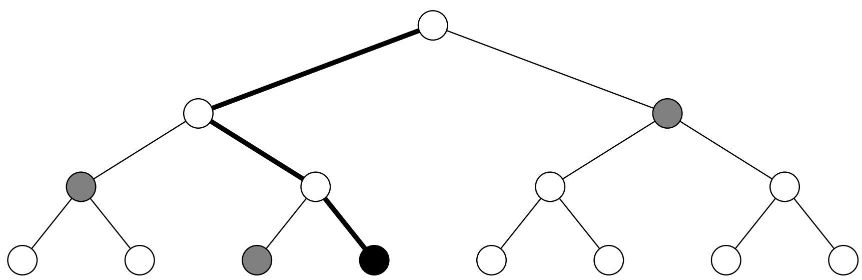

Figure 1: Merkle authentication tree

The leaves of the tree represent individual data items (or their hashes) and each inner node contains the hash of its children (e.g. concatenated or put in some data structure). The hash value at the root of the tree is the checksum depending on all the initial data items. It represents the whole data collection. In order to prove that a particular data item (for example the item colored black in Fig. 1) belongs to the collection represented by that checksum, the values in such nodes of the tree have to be provided, such that the values in nodes on the path from that particular leaf to the root of the tree can be computed. In our example, the values in the three gray nodes have to be provided. Armed with values in the black node and in gray nodes, one can recompute the value in the root and compare it with the known checksum. If these values match, then the inclusion proof is accepted.

In general, if the tree is balanced, the number of additional nodes required is logarithmic in the number of data items. This set of additional nodes will be called the *authentication path* (this name is used for historical reasons, although the nodes in the set do not actually form a path in graph theoretic sense).

6 Authentication trees with Gnutella

Currently, the files in Gnutella are downloaded in chunks. The length of chunks is quite arbitrary and there can be chunks of any length. The client has usually two chunk lengths to use — a big one like 10M for downloading from fast servers and a small one like 0.5M for downloading from slow servers. Additionally, the chunks get even smaller towards the end of the download of the file. To get efficient chunk authentication, the splitting of the file into chunks must always be done in the same and consistent way between all offerers of the file. To achieve that, we define a block size, make all the chunk sizes multiples of this block size and align the chunks to the block size. An example block size would be 0.5M or 1M.

Now that we can view the file as a sequence of fixed-size blocks, we can start building an authentication tree based on these ordered data blocks. We compute the hashes of all the blocks and create a Merkle tree of these hashes. The hash value at the root of the tree becomes the new hash of the file. This new hash is distributed to the clients with search results alongside (or instead of) the SHA-1 value that is distributed now. When the file is downloaded, the server supplies the authentication path to the root of the tree along with each downloaded block of data. The client then computes the hash of the downloaded block, adds the nodes of the authentication path and recomputes the root value (the computation process is expressed by the solid edges in Figure 1). If the resulting matches the previously published root checksum, then the chunk is authenticated, otherwise it is not.

We also need to authenticate the position of the block in the file, otherwise the server can always send the first block with correct authentication path, no matter which block the client requests. To achieve this, we can use the structure of the authentication path. We note that when recomputing the root hash from a data item, the elements of the authentication path are involved as left or right children of the intermediate nodes. The total number of blocks and the number of the current block are sufficient to establish the left/right property for each node in the authentication path. Hence, adding one L/R bit per element of the authentication path is enough to reconstruct the necessary fragment of the authentication tree and hence to confirm the position of the block in the file.

To supply the authentication path, the server must know the hash values located in the inner nodes of the tree. It would be a bad idea to recalculate them on each request since this would mean rereading the whole source file (like a 650M Linux installation CD image) on each request. So the server should cache the hash values in RAM and maybe even in some persistent cache. This increases the memory requirements of the software somewhat but is feasible, unlike the continuous rereading of files. This also puts a lower limit on the block size — the smaller blocks we have, the more nodes we have in the cached authentication tree. A mixed strategy is also possible — the server caches the hash value at the root of the authentication tree and also the values in nodes that are in several upper levels of the tree, but it recomputes the values at the lower levels whenever they are requested. To recompute the hash value at some node of the authentication tree, we only need to read the data in all leaves below that node, these leaves are not too numerous if this node is on some low level. This strategy may remove that lower bound on the size of the blocks.

7 Why add a new checksum?

Since it took more than a year to get SHA-1 hashes accepted in Gnutella network, it is easy to see that the addition of a new hash along SHA-1 would also take a lot of time. So it would be great if we could utilize the existing SHA-1 checksum and show that it depends on the specific block being in the file at a specific place.

In the computation of SHA-1 checksum of a file, this file is split to 512-bit blocks $F_1, F_2, \ldots, F_m$ (assume that the size of the file in bits is a multiple of 512, if this is not the case then use some padding). A *round function* H is used to compute 160-bit intermediate values $C_1, \ldots, C_m$ by $C_i = H(F_i, C_{i-1})$. The value C_m is the final checksum, the value C_0 is a constant fixed in the specification of SHA-1. This intermediate values could be used as the elements of authentication paths. Unfortunately, the authentication path proving that F_i is the i-th block of the file is $(C_{i-1}, F_{i+1}, F_{i+2}, \ldots, F_m)$. Hence the expected size of the authentication path would roughly be equal to half of the size of the whole file, which is too much. So in order to save the bandwidth, we really need to add a new checksum that is computed in a non-linear, efficient way (like using a tree).

8 Estimates to extra traffic

If we have 1MB block size and a 700MB file then there would be total 700 blocks. Using a binary Merkle tree, we get 10 as the height of the tree. For computations on each level of the tree we need one hash value as the input from the previous level (which is computed by the client and does not add bandwidth overhead) and one input from the authentication path (20 bytes when we use SHA-1 as the hash function). We also need one L/R bit for each element in the authentication tree. For each 1MB block of data, we get an authentication path of size $20 \cdot 10 + 10 = 210$ bytes. This is about 0.02% of the size of block, which is less than our solution quality threshold 1%. Still we need 140KB for the whole file and we should make it even smaller.

9 Optimizations

The main optimization here is to use an authentication path for a range of blocks (the whole downloaded chunk), not for each block separately. Since many consequent blocks have many values in common for their authentication paths, we can get savings from the subtrees that are fully in the chunk. The hashes for a whole subtree can be computed from the blocks themselves and so we need to have less authentication path elements in total. It turns out that we actually need only two authentication paths for the whole sequence of blocks — one for each end of that sequence [BRW02]. So the size of the authentication information gets down to $2 \cdot 20 \cdot 10 = 400$ bytes for a range of blocks. If a file is downloaded in 10 chunks, the total extra traffic makes up 4KB. This is already good enough to provide a practical solution.

We can improve the result even further because there are more efficient ways to authenticate an interval of values. Interval time-stamps solve the same problem for authenticating items that belong between two other items. So we can make use of the linking schemes used in interval time-stamping. The best known interval time-stamping schemes [Wil02] get the size of authentication paths down from $2c \log_2 n$ (from two paths of size $c \log_2 n$ where c is the length of the hash function output (20 bytes for SHA-1) and n is the number of blocks) asymptotically to about $1.44c \log_2 n$.

10 Discussion

Several open problems remain with our approach. The first of them is the handling of file tails. When most of the file has been downloaded and there are still several servers offering the file, the remaining part of the file is still downloaded from several servers which means that very small chunks of the file are retrieved from a single server. We can authenticate the last block only as a whole, so we cannot separately authenticate the sub-block-sized chunks we downloaded from different servers. The best solution seems to be to ignore the problem and gather all the pieces, check the hash of the whole block and if it does not match, discard the whole block and download the entire block again, this time from a single, randomly chosen server.

The second problem is the deployment. It took a long time to get most servers to support SHA-1 hashes and it would probably take even a longer time for this hash to be accepted for most servers (most clients do not upgrade their software without a good reason).

There are also many proposed additions to the Gnutella protocol. Many programmers and companies have developed enhancements and the process of making sure that these enhancements do not harm the network overall takes very long. There is no reason to believe that our addition enjoys any better success than the others in the queue.

Finally, it may happen that the problem disappears by itself after some time. Some of the poisoners seem to be from movie companies, they are out to discredit the file-sharing networks to scare the users away from them. We expect this problem to be short-term – it's just a technology shift that has not gotten to everyone and there are not too many regulations about this kind of behaviors yet. But there also seem to be just bad people trying poisoning for fun and it is not expected that they would go away anytime soon.

11 Acknowledgements

We thank the Tiger University Project of Estonian Information Technology Foundation for support.

References

[BRW02] Ahto Buldas, Meelis Roos, Jan Willemson. *Undeniable database queries.* In H-M. Haav, A. Kalja (Eds), Databases and Information Systems II, Selected Papers from the Fifth International Baltic Conference, BalticDB&IS 2002, Kluwer Academic Publishers, 2002.

[CS02] Andrew H. Chen, Andrew M. Schroeder. *A Modified Depensation Model for Peer to Peer Networks: Systemic Catastrophes and Other Potential Weaknesses.* University of Washington, June 13th, 2002. Available from `http://students.washington.edu/achen/papers/p2p-paper.pdf`

[Cli01] Clip2. *The Gnutella Protocol Specification v0.4 (Document Revision 1.2)*, 2001. Available from `http://www9.limewire.com/developer/gnutella_protocol_0.4.pdf`

[Co03] Bram Cohen, *Incentives Build Robustness in BitTorrent*, May 22nd, 2003. Available from `http://bitconjurer.org/BitTorrent/bittorrentecon.pdf`

[DGMS01] Premkumar T. Devanbu, Michael Gertz, Charles U. Martel, Stuart G. Stubblebine. *Authentic Third-party Data Publication.* In Proceedings of IFIP TC11/WG11.3 Fourteenth Annual Working Conference on Database Security, pp. 101–112, Kluwer Academic Publishers, 2001.

[eDon] eDonkey homepage, `http://www.edonkey2000.com/`

[Mer80] Ralph C. Merkle. *Protocols for public key cryptosystems.* In Proceedings of the 1980 IEEE Symposium on Security and Privacy, pp. 122–134, IEEE Computer Society Press, 1980.

[SHS95] *Specifications for Secure Hash Standard.* Federal Information Processing Standards Publication 180-1 (FIPS PUB 180-1), April 17th, 1995.

[Wil02] Jan Willemson. *Size-Efficient Interval Time-Stamps.* Ph.D. Thesis, Tartu University, 2002.