Building a Reliable Structured and Trusted Distributed Database on Top of Existing P2P Networks

Michael Steil <steil@in.tum.de>, Christian Hessmann <hessmann@in.tum.de>

May 23, 2004

Abstract

Existing peer-to-peer networks suffer from three main problems: no structure, bad quality of data, bad performance. This article describes a system that solves these problems by introducing a layer on top of existing P2P networks: A link database in the form of a distributed tree of index files is used, each index file being maintained by a different person. These index files are time stamped and signed, and stored in the P2P network.

1 Motivation

Existing peer-to-peer (P2P) networks like the eDonkey/eMule, SoulSeek or Gnutella network suffer from these three main problems:

1. no structure
2. bad quality of data
3. bad performance

There is no structure in the network. The database consists of a flat list of files. There are no directories to mark files that belong together, so the only way of structuring is to use archives that contain the separate files. No structure also means that the user has no way to browse the network, to see what is ”hot” or what is related. All there is, is filename search functionality.

There is no system of trust. The user cannot tell whether a file comes from a trusted source or is broken, or has even been tampered with. Viruses, worms and trojans easily spread on these networks. What is even worse, is that there is no way to remove these broken files. As long as a filename is attractive to users, the file will be downloaded - and be shared. Because of all this, the BitTorrent network doesn't even have a way to search files in the network; it relies on external databases.

The former two problems lead to a third problem: bad performance. A lot of data is shared in the form of archives, so that files that belong together can be easily downloaded. For the data to be useful, users are forced to unarchive them. The original archive is often deleted afterwards, which has the effect that it won't be shared any more. The individual files will be shared instead, which does not improve the availability of the archive. Since there is no way to find out the correct version of a file, broken versions and fakes spread equally well. So the availability of correct files is a lot lower than it could be, if everyone had and shared the correct files. Bad availability of data leads to a bad performance.

2 Introduction

All these problems can be fixed, without requiring a new network infrastructure. It can be fixed on top of existing P2P networks. The proposed system needs an underlying peer-to-peer network with the following functionality:

- There is a search function that takes parts of

the filename as a parameter and returns a list of full filenames and their unique IDs in the network.

- Files are downloaded by their unique ID. Unique IDs depend on the file contents, not the filename.

In the eDonkey/eMule network, the unique IDs would be the ”ed2k” link that consists of the file size and an MD4 hash of the file contents (as well as the filename, which is not necessary for downloading).

The basic idea of the new system is a database with a single table. Every row contains various fields that describe the file, as well as the unique ID of (i.e. the link to) the file in the underlying P2P network. So far, this is nothing special. There are many external indexing systems for common P2P networks, i.e. moderated websites that collect links to known-good files that can be downloaded off the P2P network.

But in this case, the database is distributed. The database is split into many ”index” files (”nodes”) that form a tree together. Every node is typically maintained by a different person and contains, among other things, the table description (i.e. the column headings of the fields of all rows), as well as optionally rows that are links to data files (as described above) and optionally links to more index files.

Now this sounds like a moderated link database with many moderators. But this system stores all index files in the P2P network. Some additional techniques and algorithms are needed so that authenticity can be verified, and files can be updated: All links to child nodes include the PGP public key of the author of the child node. The author’s public key fingerprint and the creation time of every index file is stored in the name of the index file, and the filename and the hash of the file contents are signed, and the signature is added to the filename. Some further techniques are applied to ensure the authenticity of the node, as well as that the author of the linked node agrees with being linked from a certain node, to prevent a takeover of the tree.

This leads to a ”tree of trust”: Less depth leads to higher trust - and the quality of data close to the root is usually higher.

3 Index files

Index files in the P2P network are located using their names, so all index files have a certain structure:

`P2PTOTvv_fff[...]fff_yyyy-mm-dd-r_sss[...]sss.txt.gz`

`vv` is the version number of the protocol. This will be ”01” for the first version. `fff[...]fff` is the public key fingerprint of the author of this node, with spaces omitted. `yyyy-mm-dd` is the date of the file. `r` is the revision of the file on this date. `sss[...]sss` is the public key signature (prologue/epilogue and newlines omitted).

The PGP signature is applied to the concatenation of the filename (up to and including the revision field) and the unique ID of the file (i.e. the hash of the file).

All index files are ASCII CSV (comma separated value) files that have been gzip-compressed. Optionally, index files can be PGP-encrypted using the author’s private key, i.e. they can be decrypted using the known public-key. In this case, the filename ends with .gpg instead of .gz, and the file does not get compressed with gzip before encryption.

The CSV fields are comma-separated, and all fields are in quotes. The contents of all fields are in C style, i.e. quotes are escaped using \ and constructs like `\n` work as expected. Lines beginning with # are comments.

The first field in every row is `"K"` (key), `"H"` (header), `"F"` (file) or `"L"` (link). There is always exactly one ”K” row in the file. It contains the public key of the author of the node in PGP format as the second, and a comment as the third field. If the file contains any ”file” rows, then there is ex-

actly one header in the file (before any ”file” row), which contains the column headings for the ”file” rows. The second field (after the `"H"`) contains the name of the the P2P system, and there needs to be at least one column with the name ”Title”. There can be any number (including 0) of file rows in the index file that form a part of the database. They have as many fields as the header. The second field (after the `"F"`) of every row is the unique ID of the file in the P2P network.

There can be any number (including 0) of link rows. Every row links to a child index file. These rows look like this:

`"L","kkk[...]kkkk", "sss[...]sss"`

`kkk[...]kkk` is the child user's public key, in PGP format. `sss[...]sss` is the signature of the (spaces omitted version of the) parent fingerprint, signed by the child (or empty) in PGP format (gpg –clearsign; parent fingerprint replaced with `%s` afterwards). Newline characters have been replaced with `\n`.

4 Algorithms

A single index file can be retrieved by searching for `P2PTOTvv_fff[...]fff_yyyy-mm` in the P2P network. `fff[...]fff` is the fingerprint of the author of the node, and `yyyy-mm` is the current year and month. The P2P network will return all revisions of this file of this month, if any, and possibly some fake files. The file with the highest date/revision and the signature intact will be downloaded. The signature can be easily verified, as the P2P client also returns the unique ID of the search results.

If no file is found, then there is probably no version from this month. Using the month before as the date, it must be tried again, until an intact file is found or a certain date is met.

All downloaded files are supposed to be shared in the network again, so that the index spreads well.

To begin with, the client needs the public key of the author of the root index from an external source. It fetches the root index and all linked files recursively. Circles must be detected; the client must stop at that point.

If the client is in ”strict” mode, only index files are accepted, if the signature entry in the link row contains a valid signature of the parent's fingerprint, signed by the child. This verifies that the child's author agrees with being linked by this node.[1] In addition, the public key in the ”K” row is compared with the public key of the child as stated in the parent's link. If not in strict mode, none of these checks will be made.

If the child index file has an incompatible header row, this is fine. Usually, a database should have identical headers in all nodes, but this makes it possible to combine different databases with a very different structure. If not all index files can be found, the database is incomplete, but still functional. In fact, it is functional right after the first ”file” row is loaded.

The client always has the complete and current index, so searching can be done offline. Every query results in a CSV row, which includes the unique ID of the file in the P2P network. The user can then choose to download files from the search results.

5 Tree of Trust

For any database in this system, there needs to be a root authority that can add any number of trusted users to its root index file. All these users can, in turn, add any number of trusted users to their index files. Every link from a parent to a child in one way needs a signature from the child to the parent of the parent's fingerprint for the children to be recognized in ”strict” mode. The contents of a single index file need not to be of a certain subtopic of the

[1] An alternative would be to make the child link back to the parent, but this way, the child can accept additional parents without updating its node.

database, the position in the tree is independent of the subtopic of the index file.

As trust cannot be strictly transitive (which is actually a good thing in this case), quality of data usually degrades at lower levels of the tree, but the number of users and the amount of data rises. The depth in the tree is equivalent to the trust level and can be used as an indicator for the quality of the data. In practice, this means that searches can optionally stop at a certain depth, or, if duplicate entries are found, the client can hide all entries other than the one with the highest trust level.

Good data can propagate up in the list, as the parent node can copy rows from children. Even users on other branches in the tree can copy entries from other lists with lower trust levels into their index file if the data is of high quality.

A maintainer of an index file can easily update it: After the file contents have been changed, the date and revision in the filename must be updated, and the filename and the unique ID (hash) must be signed again. The resulting file must be added to the P2P network.

Of course tools for converting existing structures like directories or files into the index file format as well as frontends for editing index files and their filenames are badly needed.

This tree conforms a social structure. The root index author starts a database, defines the structure and the rules of it and appoints children to help him/her. Every parent is only responsible for their children, not for the complete tree below. If a child disappoint a parent, e.g. by posting data of bad quality, the link can easily be removed. Even the an author is disappointed with a level that is not directly below, he/she can ask the child whose subtree contains the data to resolve the issue - possibly recursively. If this is not possible, the whole subtree can be dropped.

If a person decides to participate in a database, it is as easy as asking any node whether it accepts another child. The comment field in every index file can contain a message stating whether or not email contact with potential children is wanted. Usually it is not for nodes of high trust levels. New contributors are supposed to start at the bottom, and can be handed up level by level, by asking tha parent's parent. As there are usually many leaf nodes, finding a parent should not be hard. The tree should automatically remain balaced, because new contributors will typically search for leaf nodes with the highest trust levels.

6 Splitting and Linking

Since the root index file is no different than any other index file, the tree can be split anywhere. If subtrees specialize on a certain topic, or a subtree tends to contain higher quality data than the complete tree (regarding the quality per trust level ratio), this procedure makes sense. The trust level for the data just below the new root is higher, but some data might be missing.

A lower level of a subtree can link to the root of the containing tree, so that all files can be reached through this subtree. The client must detect the circle and stop there. This way, the trust levels can be resorted.

If different groups have been working on different databases on the same subject, but with the same format (i.e. "H" header rows), these can be easily linked by creating a new master index that links to the two former root index files. These will also remain independent databases. For the new database to be compatible with "strict" mode, the new root's maintainer must get his/her fingerprint signed by all children and add the signature to the links.

Databases with different headers can be linked just as well. All "file" rows contain at least the link to the file as well as the field "Title". Searching for contents of specific fields in a database like this might only return results of one child database, but searches in the "Title" field always takes all child databases into account. Alternatively, all index files of one database can be rewritten to match the format of the other. All index files of one database

will have to be updated, but all the ”payload” files will remain the same, so this requires little new amount of data in the network.

7 Potential Problems

In practice, it will happen that the maintainer of a node stops updating his/her node or that the private key for a node is lost (”dead node”). This is no problem, because the node can be replaced by a new one:

- The maintainer of the parent node of the dead node changes the link that pointed to the dead node and to the new public key and replaces the signature with one signed by the (new) maintainer.
- The (new) maintainer of the formerly dead node starts with the contents of the old index file.
- The (new) maintainer gets his/her name signed by all children of the dead node and replace the fields in the index file with the new signatures. Public keys as stored in the links of the dead node usually contain the email addresses of the owners, so it should be possible to reach all children, or else the child is dead as well.
- The filename of the new node must be fixed to include the new fingerprint and the filename and the unique ID (hash) must be signed.

Another problem can be carelessness of an author: Usually, maintainers of index files do not only extend their files with new content, but also replace links to point to files with the same content but with higher quality. The problem can arise that the new files are not distributed well - but the old files are.

In this case, the client can always go back in time and check for the same data in older versions of the index tree (which can be easily retrieved) and get the old version. This functionality can be implemented into the client and either issued automatically, or manually by the user.

The whole tree in general can be quite volatile. Any maintainer of a node can easily delete the complete tree below by updating his index file with an empty file. Even the root index file can be changed this way and disable the complete database. This power is actually intentional, but the maintainer of the root index file can also break the complete database unintentionally.

A simple workaround on the client side would be to use an older version of the tree. Clients could also use older versions of specific nodes, if the latest version is obviously broken (parse error, no links in the root index file, ...)

But there are also intentional attacks on this system. Fingerprint collisions are very unlikely. An attacker can publish an index file with the same public key of an important node, but with a broken signature. The client can easily find out whether this file is valid without fetching it. Though, it increases bandwidth for searches and slows down verification of filename/unique id pairs. Adding many fake files to the P2P network that look like the root of a popular database (but of course have no valid signature) can be seen as the Denial-of-Service attack of this system.

Another attack on the system would be to take over a database by replacing the root node with a new one that contains the same data, but is maintained by a differnet person with a different keypair, and publishing a link to this database externally to attract as many users as possible. This is unwanted, as this new user could take over the complete social network the original root author built. It is okay though to flatten the complete tree and publish it independently. This way the new database won’t be updated automatically in parallel with the original one. The idea is similar to conventions in Open Source development: It is okay to ”steal” code (copy links from one database to another), it is okay to fork a project (flatten the tree and create a new database), but it is not okay to

take over the project lead (take over the root node). The system already addresses this potential problem: All children must sign the fingerprint of the parent, and the parent must include this signature in the link to the child. A new root would have to get all signatures of the old root's children, as described above.

8 More Effects

A positive effect of this system is that if it spreads and is used by many people, even the underlying network gets better, that is, even for users that don't use this system.

All users of this system spread only files that are linked by the database. If the percentage of this people among the users of the underlying P2P network is significant, then important and verified files will be better distributed, and fakes and broken files will be a lot worse distributed. This makes the quality of the complete system better, and also improves the performance of the P2P network, as less bandwidth is wasted with fakes and broken files.

9 Further thoughts

The features of a filesystem tree can be seen as a subset of those of a database table. A table can be sorted by any column, while a file system is static. A table with a fixed order of columns is equivalent with a filesystem tree.

This way, a database in this system can represent any filesystem tree. The "Title" field would contain the filename, and the other fields would contain the directory names of the first level, the second level and so on. This way, the system can be used for maintaining a website that lives in the P2P network. A special proxy would have to convert this system into HTTP. A website like this has no problems with heavy load, cannot be DOS attacked, and includes a sophisticated system of edit permissions.

If the P2P network is used locally, on a big number of machines, this system can be used as a distributed, fault tolerant filesystem with built-in history functionality. Performance can be increased simply by adding more machines, the fauilure of machines is not critical, changes to the complete filesystem can be atomic (by releasing a new version of the root index), old versions of the complete database can still be retrieved, and updates are authorized via public key cryptography.

10 Conclusion

The suggested system is supposed to solve common problems of peer-to-peer networks and improve their overall data quality by using a distributed index stored in the P2P network that is maintained by different people. As it builds on top of an existing P2P network, and the protocols and file formats are relatively easy, this solution should be relatively easy to implement, and the resulting application is supposed to be relatively small.

We are working on a proof-of-concept implementation of this system, built on top of the eDonkey/eMule network in the Perl programming language. This project can be found at: `http://tot.sourceforge.net/`.