Data Deduplication in BitTorrent

João Pedro Amaral Nunes

October 14, 2013

Abstract

BitTorrent is the most used P2P file sharing platform today, with hundreds of millions of files shared. The system works better the more sources one has to download from. However, BitTorrent only allows for users to find sources inside the same torrent, even if a file (or pieces of it) are located in many other torrents as well.

Using Deduplication techniques, we develop a system that allows users to detect and take advantage of those extra sources, and measure its efficacy at finding sources with a low overhead for the user.
Keywords:*P2P, Peer-to-peer, BitTorrent, File-sharing, Deduplication*

1 Introduction

Peer to Peer (P2P) file sharing applications are very popular, composing almost 50% of the upstream data usage in the US and 10 to 20% of the total internet traffic across fixed network connections, which is a monthly average of 6.15GB per subscriber on the US alone [14]. BitTorrent is the largest of the peer-to-peer file distribution networks currently used [14], with over 52 million peers connected to over 13 million active torrents, sharing almost 280 million files, totaling over 15EB of information[1].

In P2P file sharing networks [1], having more sources for a file (or pieces of a file) has multiple advantages. Faster download speeds is one such advantage. A user with limits regarding international traffic may like to find sources on the same country. On locations with limited internet access, finding sources in peers from a local area network may be the difference between finishing a download or not. So, being able to find all available sources for a file would be advantageous for anyone downloading it.

BitTorrent can only locate sources for whole torrents [3]. However, some files are present in many torrents, and some files even share a part of their content in common.

Ferreira, in his thesis [5], cites 8.7% of the examined torrents contained over 75% of its content in common. Pucha et al. [12] found the median percentage of data shared between MP3 files (its *similarity*) in the eDonkey and Overnet P2P networks to be of 99.6%, with a median of 25 files in a set of 3725 MP3. Practices such as bundling [8] and repackaging [5] create even more duplicate sources for a file, which combined with the above findings, mean there is an immense quantity of potential sources that is currently not exploited .

Our objective is to allow the user to find many more sources than BitTorrent currently allows, while minimizing overhead on the user. We also need the system to be realistically implementable at the scale of BitTorrent.

In order to exploit those extra sources, the file sharing system need to be able to identify and locate redundant data. To do it, we turn to the field of data deduplication.

Using an algorithm from the Compare-by-Hash family [6], there is a way of detecting duplicates across the torrent network. If we have an unique identifier for the chunks of a file, we can use that identifier to find similar or identical files through the network.

However, compare-by-hash algorithms have a problem. Small chunks allow for greater redundancy detection, but increase the size of the metadata. Large chunks reduce the amount of redundancy that can be found, but reduce the size of the metadata.

[1]Information extracted from the torrent indexing site 'IsoHunt.com' in September 2013.

In this document, we will present another solution. First, using a new approach at fingerprint storage, multi-level fingerprints, we create a type of metadata that can be small in size for the end user, yet capable of detecting sources. Then, we designed a set of services that allow our solution to find and expose redundancy to the user, all the while keeping the user overhead at a minimum as well as stay viable on the real world.

2 Related Work

To the best of our knowledge, there is no solution that can detect and find these duplicates implemented in the commercially available BitTorrent applications. SET [12] is a P2P file sharing system capable of exploiting that similarity.

SET uses the concept of a handprint, an ordered finite set of a file chunk's fingerprints. A handprint is an heuristic to determine if two files are similar to each other. Two files are considered similar if both file handprints share at least one fingerprint in common. As the amount of fingerprints used to form a handprint increases, so does its capacity to detect similarity between files. This capacity to detect similarity grows logarithmically with the amount of fingerprints used, with 30 fingerprints allowing to detect files with at least 5/

The SET approach, however, has some fundamental flaws. The handprint system offers no guarantee of which chunks are similar between both files. If a SET user is looking for more sources for a group of chunks in his file, there are no guarantees that the file sources provided by SET would contain the chunks that he needs. That may cause the user to waste bandwidth and time on something that brings him no benefit whatsoever.

Another flaw is in SET's metadata size. SET's metadata has about 0.1% of the size of a file on average. That means a 1MB metadata file for a 1GB file. In BitTorrent, 1GB files have a metadata of close to 5KB in size, over 200 times smaller than SET's metadata.

It is still reasonable that a cilent would not mind downloading SET's metadata once. However, a client may have to download multiple file metadata to make use of SET's service. When a handprint present a match, it returns a list of files that are similar up to the designed percentage. The client must then download the metadata for each of those files in order to find which chunks are identical in order to download them. If we had 200 variants of our 1GB file, each with 1MB worth of metadata, the user would have to download 200MB worth of metadata, 1/5th the size of the original file.

3 Architecture

Our system is composed by several distinct pieces of software, called *Services*. There are three services in this architecture:

- The *insertion service* is responsible for finding extra sources, for both chunks and files.
- The *lookup service* is responsible for keeping the information found about extra sources, and deliver them to any process that requests it.
- The *maintenance service* is responsible for making sure the lookup and insertion services are working correctly, and that the information in both these services is up-to-date and fresh.

Only the insertion and lookup services are contacted by the users.

Our system is built in order to deliver *sources* of duplicate pieces of data to anyone that requests them. Sources are ID/offset pairs. What the ID is may vary depending on the type of source. For a file source, the ID is the torrent's infoHash. For chunks inside a file, the ID is the whole file hash, the *file ID*. The offset is the location inside the container where the block of data we wish to find is located. For a file, the offset would be where inside the torrent that file begins. For a chunk, the offset is where in the file the chunk begins.

A user wishing to upload a file into our system (an *uploader*) starts by creating the metadata for that file. Then, it submits that metadata to the insertion service for processing. The insertion service detects redundancy and stores the information about found duplicates in the lookup service.

Further on, a user looking for extra sources (a *peer*) contacts the lookup service providing the fingerprint for the chunks that needs to be found, and the lookup service replies with a list of sources found so far for any chunk that had a duplicate available.

3.1 Metadata

In order to start using the system, an uploader must create the metadata for the file the uploader wishes to submit to the system.

In order to keep the metadata the peer must download small, yet provide the redundancy detection of small-sized chunks, we developed a multi-level fingerprint structure, where each level provides a different level of resolution. There were previous work done in multi-level redundancy detection ([4], [7]), but our work takes a different approach.

The structure representing the metadata is a Merkle tree[9], where each node represents a chunk of data. The root of that tree is the whole file node, containing the file ID. Each level of the tree represents a *level range*. The level range is a pair of values that identify the smallest and biggest possible value for the size of a chunk in that specified level. The level ranges get smaller as the depth of the tree increases.

Nodes in the tree represent a chunk of a size inside the level range. The node contains the fingerprint for that chunk, the size of the chunk and the offset of that chunk inside its parent. The children of that node are the chunks resulting from splitting that node into chunks of a size inside the next level range.

In order to better understand this structure, we provide an example in the figure below.

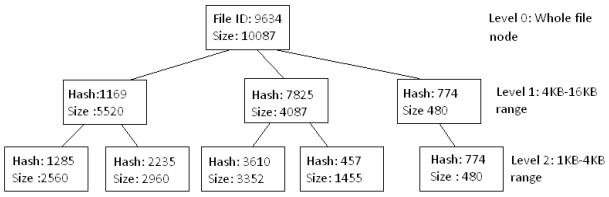


Figure 1: Example of a small tree

The chunks used by our system are variable-sized chunks [10] with a boundary determined by using a Rabin fingerprinting function [13]. The fingerprints used by our system are the SHA-1 hashes of the chunk, which are relatively safe to use as identifiers [2].

Using this metadata structure, a peer is able to download a small metadata structure at first, but then download sub-trees for select chunks the peer wishes to find more sources for. That way, the peer only downloads the amount of metadata actually needed, minimizing the space taken.

3.2 Lookup service

The lookup service is responsible for storing the extra sources found by the deduplication process. The lookup service is also responsible for replying to any peer or service that asks for extra sources.

From a peer perspective, the lookup service receives a request containing either a file ID or chunk fingerprint for which the peer wishes to find more sources, and replies with a list of sources for that data. If no sources were found, the lookup service replies with an empty list.

Besides replying to requests, the lookup service must also store the sources found by the deduplication process. The sources are provided by the the insertion service, and the lookup service needs only keep this data safely stored.

In order to store the sources, the lookup service contains two tables. The *file lookup table* stores sources that contain found files. The *chunk lookup table* stores sources for chunks found duplicate by the system.

The file lookup table simply stores the file ID and a list of sources.

The chunk lookup table contains the fingerprint of a chunk,the list of sources of where to find that chunk in a file, called a *source list*, and a list of sources of where to find that chunk inside another chunk, called a *parent list*. When looking for chunk sources, the lookup service returns the source list for the chunk, as well as all the sources for the chunks in the parent list. This method of retrieving sources is called a *parent chain*, and reduces the size of the chunk lookup table by only storing sources on the biggest duplicate chunk found.

3.3 Insertion Service

The insertion service is responsible for finding duplicate data and inserting the results into the lookup service.

The insertion table contains two tables, a *file insertion table* that contains information about files that were submitted to the insertion service,and a *node insertion table*, that contains the fingerprints of nodes submitted to the insertion service that have no duplicates so far .

The insertion process starts when an uploader decides to submit a file. The uploader first sends to the

insertion service the metadata root node, with the source for that file.

After receiving the root node and source, the insertion service will check if the file was already present. If the file was present, the insertion service sends the new source to the lookup service and finishes the insertion. Otherwise, the insertion service will require more metadata from the uploader., may require the metadata for insertion, if the file was not yet present in the system.

The insertion service may request the uploader sends the entire metadata at once, or that the upoader sends the metadata one level at a time. In either case, for each remaining level, the insertion service looks up the chunks in the lookup service. Chunks found in the lookup service have a new source added to their lookup table entry.

For each chunk that was not found, the insertion service looks in the node insertion table for it. If found, the chunk is inserted into the lookup table, with the source from the table and the source from the file being examined.

Finally, each remaining chunk not found is stored in the node insertion table.

Because the amount of data in the BitTorrent network is so large, it is almost impossible to store all nodes in a single system. As such, there will come a time when old nodes without duplicates have to be deleted to make room for new ones. However, we have no way of knowing which nodes are, in fact, useless.

In order to decide which nodes to delete, we take inspiration from SET's handprints. The way the nodes are chosen for deletion is explained in section 3.4.

In a second phase of the insertion algorithm, for each file that shared at least one chunk with the one being inserted, we create a set containing all the chunk fingerprints bigger than the handprint end for that file. If there are any fingerprints in that set, the insertion service contacts someone that has the metadata for the file being checked, and any fingerprint from the set that is found is inserted into the Lookup service.

Finally, for all newly inserted nodes, the insertion service inserts the entire sub-tree into the lookup service.

3.4 Maintenance Service

The maintenance service has two major functions.

The first major function is to make sure the sources reported in the lookup file table still exist.

The other major function is to make sure the insertion tables do not reach limit capacity.

Besides the major functions, the maintenance service also performs smaller jobs, such as checking the system for major faults, such as the complete erasure of the tables of one of the services.

The maintenance service periodically checks the files to see if their sources still exist. After a period of a source being unreachable, the source is removed from the lookup service. If a file has no more sources, the file entry is removed.

The removal of the file entry causes a purge of that file from all chunk sources and from all entries in the node insert table. The file insert table entry for that file is also deleted. Then, for every chunk affected in the chunk lookup table, if they have no sources, they are erased, and if they have only one source, they are moved back to the insertion table.

We require to erase nodes from the node insertion table. To choose which nodes to delete, we took inspiration from SET.

Handprints provide similarity detection between files up to a certain percentage, with only a small amount of chunks. However, using handprints introduces some problems, such as not being able to detect the location of specific chunks and needing to download the entire metadata to discover which chunks were in common.

Using a handprint will increase the message overhead and time taken for the insertion process. Also, some chunks may become impossible to find after deleted, if some of the metadata is lost for all peers. There is also the possibility that some files share only chunks greater than handprint end, and those duplicates would be impossible to detect.

For all the above reasons, we would like to use this solution as sparingly as possible, and not blindly apply it to every file present in the system.

The maintenance service will delete nodes in reverse order, starting with the biggest fingerprint. We then store the biggest fingerprint still remaining for that file in the file insert table. The files selected to go through the cleanup process will be the files that were least recently accessed, that is, files that did not have a chunk read in the insertion process for the longest time.

3.4.1 BitTorrent extension

Up until now, the system definition allows our system to work for any P2P files haring network. However, for a peer to reach the system, we need some client that can connect both to our system and to the P2P file sharing network.

As the objective of this project is to detect and exploit redundancy in the BitTorrent network, we have designed a way for BitTorrent clients to fully take advantage of our system, by the means of a BitTorrent expansion [11].

The BitTorrent expansion allows for messages to request a list of known duplicates from another peer. This may reduce the workload on the lookup service. It also allows for peers to exchange the metadata pieces, allowing them to download sub-trees as they need them. Another message allows for peers to request from a peer that has a chunk which sub-tree has been lost to recreate it.

4 Evaluation

With this evaluation, there are several questions we would like to answer. How many additional sources can we find for each of our given files? What is the expected time for inserting a file into the service? What is the overhead produced by the services? Can we, at least, provide as many sources as SET?

We define the popularity of a torrent as the total number of seeders, with a torrent with many seeders being more popular than one with few seeders.

The number of whole file sources is the number of distinct torrents which contain the file. To simplify our evaluation, this number is used as the total number of copies of a file that exist in the network.

The number of chunk sources is the number of files from where one can obtain a given chunk. These sources may be from inside the file itself (internal redundancy). A chunk is worth a percentage of the file it is in, which is given by dividing the size the chunk by the total size of the file. This is called the *chunk percentage*.

The percentage of chunk sources is given by the equation

$$p = \frac{\sum size_{C_i} * fileSource_{C_i}}{fileSize} * 100$$

for every chunk C_i that that composes the file. The size of a chunk C_i is given by $size_{C_i}$, $fileSource_{C_i}$ is the total number of files where C_i was found.

The total availability of the file adds to the above equation $100 * wholeFileSources$, and represents the percentage of the file available in the network. The availability is always bigger than 100 for every file.

4.1 Experimental methodology

Used data set

In order to evaluate this system, we first downloaded the top 10 most popular torrents from the torrent indexing site "piratebay.se" audio section, which included music in mp3 and m4a formats, and videos in the mpeg format. For each of those top 10 popular torrents, we searched for other torrents that shared the same artist and name, and downloaded a maximum of 10 from that list. Those other downloads, by comparison, usually had one tenth to one hundredth of the original torrent seeders, being much less popular.

The total number of torrents downloaded and analyzed was 62, with a total of 1154 files, of which 956 were distinct, and 198 files had at least one copy. Among these 198 files, one file had 7 copies of itself, 32 files had 3 copies, 4 files had 2 copies and 90 files had only one other copy. This means that one of the files had 8 whole file sources, 32 files had 4 whole file sources, 4 files had 3 whole file sources and 90 files had 2 whole file sources.

Experimental procedure

We generate the metadata for each downloaded file using our multi-level metadata generation algorithm. We then proceeded to insert them into our system, using a test client that activated the insertion service. During the insertion process, we collected several measures, namely the time it takes to insert a file and the associated network overhead.

After all files were inserted into the system, we used a test client that measured the number of sources found per file. The results are presented in the above described percentage of sources found, and distinct sources found. We also present the time it took to find all sources, assuming a greedy client that wants to find all sources for all chunks.

Finally, we compare the results obtained with the number of sources found using SET's algorithm.

Experiment setup

All tests were performed on a single machine, with an Intel i7 3610 CPU at 2.3GHz. The computer ran on Windows 7 64-bit operating system, using Java Runtime Environment 7u16 with 2GB maximum heap size. Both Insertion and Lookup tables were stored in MySQL 5.6, inside one Samsung 830 SSD. All the tests were run only once, due to time constraints.

4.2 Insertion evaluation

The first step of insertion testing was generating the metadata. For each downloaded torrent, we created one set of metadata per file, with at most 9 levels, starting at 1024 bytes and ending at 64MB.

The average size of the metadata was 2.89% of the total file size, with an standard deviation of 1.14%. No file over 2KB had more than 7% of its size in metadata. The time to build the metadata file grows linearly with the size of the file analyzed, with an average of 0.43ms/KB and a standard deviation of little over 0.01, for any file over the size of 50KB.[2]

After the metadata was generated, we started inserting it into the system. We grouped up the files by the torrent they belonged to, and inserted one torrent at a time. As inserting a torrent means to simply insert each of its files in order, the results were simply the sum of the results of the composing files data. We will, therefore, ignore the whole torrents insertion data, and focus on the files only.

We measured the time and message overhead of the insertion process. Because our file selection contains files of vastly different sizes, insertion time and message overhead would also vary greatly with each files, making it difficult to examine the data. As such, we will usually present ratios of time and sizes, by dividing those values by the number of nodes of a given file.

We can divide the files inserted into two main categories: Duplicates and originals. Originals are files that either have no whole-file duplicate or that were the first of the duplicates to be inserted. Duplicates are all files where at least one copy was already present in the file lookup table.

[2]We did not include files under 50 KB due to their ratio being vastly bigger due to rounding errors in the time taken.

4.2.1 Duplicate insertion

The insertion of duplicate files shows us something we already expected from the algorithm. Duplicate files are looked up and inserted very fast, and have almost the same message overhead across multiple files. This happens because each duplicate insertion only takes one lookup and exchanges only one node, the file node. This table presents the value obtained for the times as milliseconds and message overhead as bytes, not as ratios. The values are nearly constant for all duplicates inserted, a total of 198 files.

	Average	σ	Max	Min
Time	6.87	2.34	20	4
Message Overhead	206.19	1.93	209	199

4.2.2 Original file insertion

The insertion of original files involves a different part of the algorithm. The number of nodes now heavily influences the time and message overhead for the file insertion. By ordering the files by the number of nodes, we can see the trend as the number of nodes grow for the algorithm to stabilize the ratios of message overhead and time.

The Figure 2 show a stabilization of the ratios as we reach files with over 50 nodes. We separate the files into two groups: files with less than 50 nodes (first 168 files) and files with at least 50 nodes (785 files).

The ratio between nodes and time at the lower group varies greatly, with a maximum registered of 23 ms/node and a minimum of 1.69ms/node.

For the group with over 50 nodes, the time taken is quite stable, with an average of 0.99 ms/node and a standard deviation of 0.6.

For our largest file, a 2.5 GB file with 2 million nodes in its corresponding hash tree, the insertion took nearly 20 minutes. We consider 20 minutes to insert a 2.5 GB file is an acceptable time for an uploading peer, even though the system was designed to be easy to use for the normal peer.

Looking back at the message overhead graphic, it is noticeable that after 50 nodes, some files have a much lower ratio than the average. The files that break the

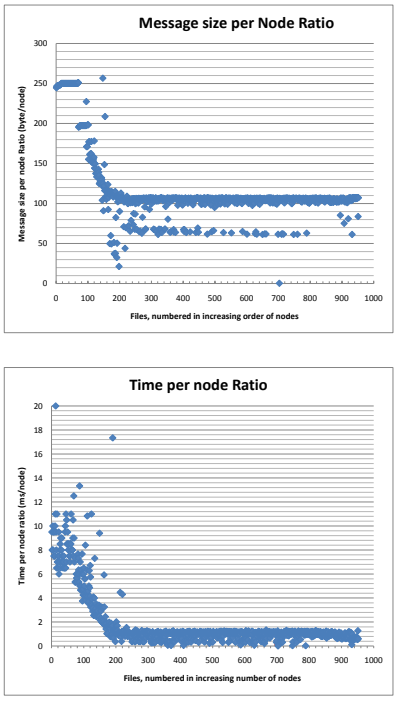


Figure 2: Time and Message size ratios for all original files

average found some duplicates in higher levels of the tree already present at the Chunk Lookup table.

When chunks are found that way, the algorithm does not need to request that entire sub-tree, and that is evident in the reduced message overhead, with the average being 72.51 B/node, a reduction in nearly 30 B/node. This proves that our level system is useful in reducing the network overhead from transferring metadata, as long as some higher-level nodes are detected.

4.3 Lookup testing

After finishing the insertion, we used another test client to lookup every file, in order to find out how many sources were found.

The client would open the metadata file and, for each level, would request from the Lookup Service all chunk sources found. When the client finishes looking up the entire file, the client would then gather several statistics, such as the number of files sources found, the number of chunks found in common between files, the internal redundancy found in each file, the percentage of the file available in the network (by adding up all existing sources) and the time and message overhead for the system usage. We will begin by checking the number of file sources found against the number of sources found using the SET method.

4.3.1 Number of file sources found

Being the closest related work available for testing, we implemented a simple version of SET for comparison with our work. We measured the number of sources found by both systems, as well as the number of fingerprints required to be stored to keep the system functioning. We used the second to last level of our metadata structure (the 4KB-16KB range level) as the metadata for SET. We used the first 30 ordered hashes from that level as the handprint. This causes SET to detect files up to 5% similar.

We implemented a client that first inserted the handprints of every file into a table. Then, for each file, the client looked up the handprint from that same table. We then compared the number of file sources found against the results obtained with our algorithm. Finally, we compared the space taken by useful data from SET memory versus our own from the Lookup Table.

We extracted the number of files that have at least one chunk in common from our source results. 196 files did not have a single external redundant chunk, and are not considered for this test. The remaining 760 files were analyzed, and the results are presented in the figure 3.

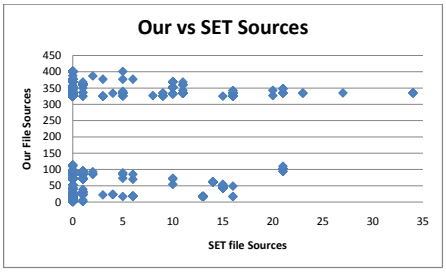


Figure 3: Files detected using SET vs. using our approach. The SET file sources scale had to be adjusted in order for SET results to show on the graphic

There are a large number of sources for files we have detected that SET has not.

SET only detected 294 similar files out of the 760 files with shared similarity. SET failed to detect 50 files within the range of similarity it was supposed to detect, mostly because the nodes in common were detected at a lower range (1KB-4KB range). There was no case SET detected we were unable to detect, which was expected as determined before in the Section 3.3. There was no case where SET detected the same amount of sources our approach did.

From the graph, we can clearly distinguish between two groups of results. One set of files shared between 400 to 300 file with some similarity. The second group shares between 120 files to one file with some similarity. The reason for this file division is due to the files in the 400-300 range sharing a group of very common chunks with each other (padding chunks).

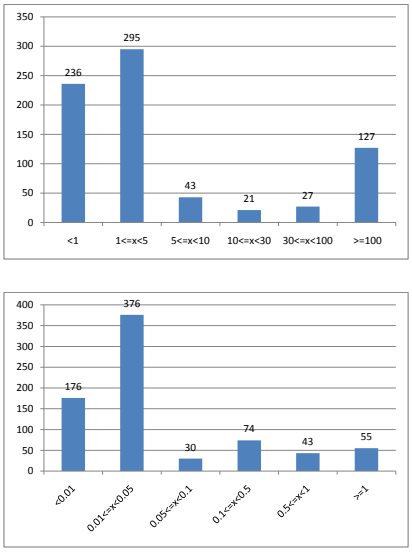


Figure 4: Maximum and minimum similarity between any pair of files. The x axis represents a percentage interval, while the y axis represents the total number of files

4.3.2 Found sources

We began by calculating the maximum and minimum similarity between files. To do this, we calculated the percentage of sources found for each chunk when considering only one other file as the extra source. We ignored whole-file copies for this evaluation. The results are presented in the figure below:

As we can see in the upper graphic, most files that had duplicate chunks never shared more than 5% of their contents with another file, totaling 531 of the 753 files. Out of the remaining files, most files (127) have more than 100% of themselves in common with another file. What this means is that there is a file out there that contains this file, with some internally redundant chunks also shared by these files.

On the bottom graphic, we see that most files shared less than 0.05% of their content with another file. While this is very little, it still represents 500KB of data on a 1GB file form an extra source. We can also see that only 55 files had at least 1% or more share content between them. This chart clearly shows there is much similarity to be exploited on the very small similarity range. These small pieces add up, and the result is an increased availability as presented in the next graphic.

Figure 5 tells us what is the total availability of a file in our system. The majority of files (491 in total) fall in the range between 100 and 150% availability, with 80 files fitting in the 130 to 150% range. There are 269 files with over 150% availability, with 218 files having more than double the availability from different files using our system.

All the above results support the claims that there is much redundancy between files in P2P networks, and that much of that redundancy happens in very small similarity ranges. The result also show that adding all those small chunks together increases the total availability of a file by a significant amounts.

4.3.3 Time and Message overhead

Another important aspect of our system is the time it takes to lookup a chunk, and the overhead consumed by that action. Our lookup test is the worst case scenario, where the user looks up the entire tree. This is not the typical, expected interaction.

Once again both functions show a stabilization of the time and message overhead ratios as we reach files with over 50 nodes. As such, we will examine the less than 50 nodes case separately from the greater than 50 nodes case.

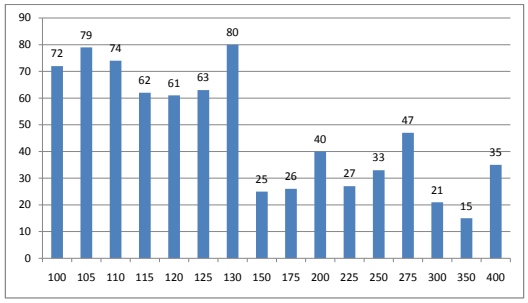


Figure 5: Total Availability of a file, not counting extra whole file sources

The full tree lookup takes, on the average case for larger trees, 17% the time of the insertion, 0.17ms/node, with a maximum registered of 1.34 on the top of the tree. The message overhead, however, is much more spread out, although also smaller. The reason for these wildly varying values is due to the number of sources found, which can be much greater than the number of nodes in a good run.

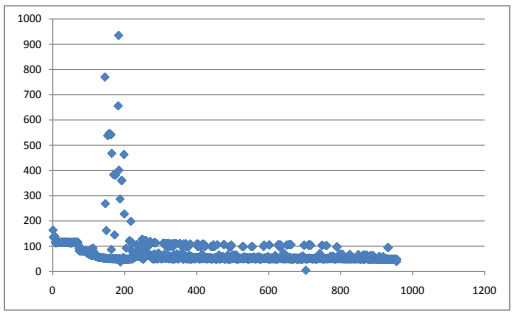
The following are the graphics that show the system performance in the worst case.

Another important note is that the time it takes to lookup a file is always increased if more sources are found. It took little less than 8 minutes (455908 ms) to lookup the entire tree for the largest file, that has a 1.04% extra sources. However, it took almost double that (under 12 minutes, 701847ms) to lookup the second largest file, that had 28.43% extra sources.

Overall both this and the Insertion Service tests show that our system is more adequate for files with more than 50 nodes, that is, files that are over 60KB in size. These tests also show that the system is capable of accurately identifying a large quantity of sources for many common chunks, even across many files that have little in common.

4.3.4 Space occupied

The total size of the node lookup table is 122MB in disk, with only 28MB actually containing table information. Considering the working set has 39 GB, this means the ratio between data processed and space occupied by the nodes was 0.07% at least, and 0.3%



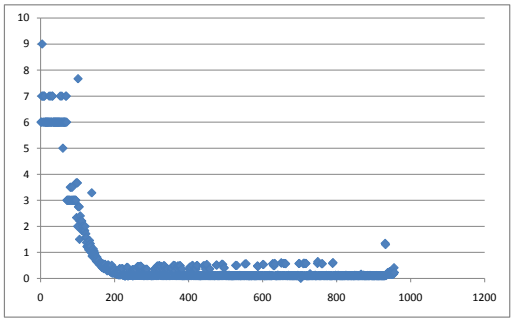


Figure 6: Time and Message size ratios for all original files

at most.

The SET table was setup in memory, and took about 5MB in Java's virtual memory, with an estimate 1MB of actual data content.

SET table had 20070 empty nodes out of 23765 total. These empty nodes are only potentially useful.

When balancing both solutions, ours offers greater accuracy and ease of use in exchange for disk space. SET is fast and small, but requires the user to download large metadata files and find common chunks on his own, and is not capable of identifying as many sources as we do.

5 Conclusions

When we began this project, we started with a simple question: "Would it be possible to detect similar data across BitTorrent and exploit that to find additional sources?" We knew that whichever solution we brought up had to be capable of dealing with the massive amount of information in the BitTorrent

network, as well as be simple to use and accurate.

In the end, we proposed a novel approach composed of a group of services: the insertion service, that found and gathered the redundant data; the lookup service, that provided the users the information about the sources and was responsible for keeping it stored; the maintenance service, responsible for keeping the other two services in working order.

We also created our own type of metadata, a multi-level fingerprint tree that brings together the benefits of small chunk redundancy detection, yet can be distributed in a much smaller size to the users.

We implemented and evaluated a version of our three services, as well as the metadata structure.

We have shown that it was capable of detecting similarity between files even when the similarity was as low as 0.01%. Our system detected at least as many file sources as SET, finding between 10 to 100 times more file sources than SET. We also shown that many files share very little in common, but adding the common chunks together created a much larger availability for the file. In over 1/3 of the tested files, that availability increased by 50%.

Our lookup service spent 28MB in useful data reporting on 39GB of unique files, reporting nearly 498000 different nodes. It could reply to the client in an average time of 0.1ms per node, spending an average of 145 bytes of message overhead per node accessed. The time taken to insert per node is close to 1ms, and spends an average of 101 bytes overhead per node.

All the above results obtained meet our requirements of a low overhead on the user, and implementing the system in a way that could scale to the entirety of BitTorrent.

References

[1] S. Androutsellis-Theotokis and D. Spinellis. A survey of peer-to-peer content distribution technologies. *ACM Computing Surveys (CSUR)*, 36(4):335–371, 2004.

[2] J. Black. Compare-by-hash: A reasoned analysis. In *Proc. 2006 USENIX Annual Technical Conference (Boston, MA*, pages 85–90, 2006.

[3] B. Cohen. The bittorrent protocol specification, 2008.

[4] Kave Eshghi, Mark Lillibridge, Lawrence Wilcock, Guillaume Belrose, and Rycharde Hawkes. Jumbo store: providing efficient incremental upload and versioning for a utility rendering service. In *Proceedings of the 5th USENIX conference on File and Storage Technologies*, FAST '07, pages 22–22, Berkeley, CA, USA, 2007. USENIX Association.

[5] António Ferreira. Monitoring bittorrent swarms. Master's thesis, Instituto Superior Técnico, 2011.

[6] V. Henson. An analysis of compare-by-hash. In *Proceedings of the 9th Workshop on Hot Topics in Operating Systems (HotOS IX)*, pages 13–18, 2003.

[7] N. Jain, M. Dahlin, and R. Tewari. Taper: Tiered approach for eliminating redundancy in replica synchronization. In *Proceedings of the 4th conference on USENIX Conference on File and Storage Technologies*, volume 4, pages 21–21, 2005.

[8] D.S. Menasche, A.A.A. Rocha, B. Li, D. Towsley, and A. Venkataramani. Content availability and bundling in swarming systems. In *Proceedings of the 5th international conference on Emerging networking experiments and technologies*, pages 121–132. ACM, 2009.

[9] Ralph C. Merkle. A digital signature based on a conventional encryption function. In *A Conference on the Theory and Applications of Cryptographic Techniques on Advances in Cryptology*, CRYPTO '87, pages 369–378, London, UK, UK, 1988. Springer-Verlag.

[10] Athicha Muthitacharoen, Benjie Chen, and David Mazières. A low-bandwidth network file system. In *Proceedings of the eighteenth ACM symposium on Operating systems principles*, SOSP '01, pages 174–187, New York, NY, USA, 2001. ACM.

[11] A. Norberg, L. Strigeus, and G. Hazel. Bep 0010 extension protocol, 2008.

[12] Himabindu Pucha, David G. Andersen, and Michael Kaminsky. Exploiting similarity for multi-source downloads using file handprints. In *Proc. 4th USENIX NSDI*, Cambridge, MA, April 2007.

[13] M.O. Rabin. *Fingerprinting by random polynomials*. Center for Research in Computing Techn., Aiken Computation Laboratory, Univ., 1981.

[14] Sandvine Incorporated ULC. Global internet phenomena report: 2h 2012, November 2012.