DetectIt: A Peer-to-Peer Plagiarism Detection System

Elif Tosun, Ben Wellington
Department of Computer Science
Courant Institute, NYU

May 7, 2003

Abstract

In today's society, plagiarism is a big issue in academic and professional environments. Many copy detection systems have been proposed to solve this problem, but they fall short in providing a fault tolerant, scalable solution that respects intellectual property. In this paper, we present "DetectIt", a peer-to-peer scalable plagiarism detection system. It is built on top of the Tapestry framework and uses a fingerprinting technique to detect parts of documents that are similar. Our results show that average the turnaround time is less than .75 seconds for a query document of 10,000 characters.

1 Introduction

In today's wired society, the act of plagiarism takes less effort then it did just decades ago. A plagiarist doesn't even have to go through the effort of rewriting. With a simple "copy and paste" on a computer or on the Internet, it is possible to copy entire documents in just a few mouse clicks. In fact, recent studies show that in some college classes, the average plagiarism rate is as high as 17 % [2]

Currently, there are several pay sites on-line that offer plagiarism detection. We have found several problems with these sites. Firstly, these are fee-based services! This makes the widespread use of these services unattractive. Secondly, if different professors in one university use differing services, they will not detect copies within the same school system. Lastly, storing students' papers in an on-line database could be considered a violation of a student's intellectual property, as the documents are essentially handed over to a third party.

For applications such as plagiarism detection, the ideal solution would involve keeping track of all documents written all over the world so that when a new submission is made, one could check the similarities between the query document and all the other documents in the database. The extent of the ideal solution is due of the nature of the problem. That is, the more the users, the more effective the results are. However, the amount of space required and the time for searching such a huge database as well as maintaining such a database with many updates a day makes this solution impossible to be employed by some software running on a single machine. Another problem with this approach is that it leaves the main server open to attack, since that server constitutes a single point of failure. This is actually one of the problems with most of the current systems. The above reasons make this problem very suitable for a distributed system based solution, where each user runs a copy of the distributed software that keeps track of all the papers that have been entered into the system by all the users, possibly worldwide.

DetectIt is a peer-to-peer system that approaches the problem in this direction. It is built on top of the Tapestry API, which is made to be fault tolerant over a wide area of distribution. This approach is attractive as there is no central database storing students' papers. The teachers themselves have the only copy of the paper. However, some crucial information that would be used in similarity detection, called *fingerprints* (see Section 3) is stored in the system that every user has access to. When a paper is deemed to be highly similar to another one that is in the system, the user is given a professors id, and a paper id. He/she can then contact the professor that originally submitted the paper, and retrieve the document using the paper id.

2 Background

Our contribution is an implementation of a new application to an existing system which is explained in full detail in [15]. Although the application to plagiarism detection has been strongly suggested in [15] we are not aware of another implementation of a fully decentralized peer-to-peer plagiarism/copy detection system. In this section, we briefly explain the underlying structures that our application is based on.

The design of this system consists of several layers. The lowest layer in the hierarchy is Tapestry, a peer-to-peer overlay location and routing infrastructure described in [14]. Above that is the "Approximate Text Addressing" (ATA) layer described in [15]. Our application DetectIt is implemented on top of this layer. The following sections provide brief explanations of each layer to make the reader familiar with the background. For further details the reader is referred to the papers cited in each section.

2.1 Tapestry

Tapestry is one of many peer-to-peer overlay location and routing infrastructures available today such as Pastry [8], Chord [12] and CAN [7]. Our design choice favored Tapestry due to the fact that a very crucial layer of our system, the "Approximate Text Addressing" layer extends the implementation of Tapestry, although ATA could be built on other such infrastructures with few modifications as suggested in [15].

The idea behind Tapestry routing is point-to-point links based on matching digits in the destination ID and the current node ID. This is maintained through the use of routing tables at each node called "neighborhood maps" that consists of several levels. Each level corresponds to a matching suffix up to digit position. As the example in [14] suggests, the 9th entry in the 4th level of some node with ID 325AE is the closest node in network distance that ends in 95AE. This organization of point-to-point linking supports a maximum of logarithmic number of hops. As a side note, to support dynamic node insertion each node also keeps a back-pointer list that points to nodes that it is referred to as a neighbor.

Tapestry locates objects stored in the system in the following way: When Tapestry starts up each server "publishes" all objects that are stored within that server. Publishing consists of routing a message to a root of the object. Tapestry maintains several roots per object in order to support fault tolerance. Roots are defined by some hash function called on the object ID and a global sequence of "salt" values to allow multiple roots. At each hop of this publish message, the server location information is stored as a mapping in the form <obj ID, serv ID>. Thus, during a location query, a message for a given object is first directed to one of its roots. If at any hop towards the root a mapping of the above form is found, the message gets redirected - this time towards the server of the object.

A tapestry network is first initiated by a federation node and a set of static nodes. As a side note, the term "node" here can be a different machine or different ports on the same machine. The federation node awaits to be contacted by a number of static nodes defined in a configuration file. Once it hears the signal from all static nodes, it sends the information about all other nodes to each of the participating nodes. Then the nodes sync among themselves, and the federation node is of no use any longer. From this point on, any other Tapestry node that defines itself as a "dynamic" node can insert itself into the network based on the dynamic insertion algorithms explained in [14] as long as it knows at least one "gateway node" in the network. Note that any node in the system can act as a gateway.

2.2 ATA (Approximate Text Addressing)

In infrastructures such as Tapestry, one cannot perform a search based on general characteristics of an object. In Tapestry, each object receives a globally unique identifier(GUID) and the search is conducted on the set of GUIDs. However, in our case instead of just a name for an object, we want to perform a search based on the contents of a given query document. Although this concept is not supported in Tapestry, there exists a middle layer called "Approximate Distribute Object Location and Routing (ADOLR)" implemented by Zhou et al. [15] that performs the necessary translation between what the application needs and what Tapestry supports. Although ADOLR is a general purpose layer, the authors of [15] provide the implementation for a specific use, namely "Approximate Text Addressing (ATA)" that allows searches on properties of similar documents, which is exactly what our application needs. Here, we briefly explain the mechanism behind ATA.

ATA allows searches within the system to be performed on a set of values rather than just one GUID. The set of values correspond to the fingerprints that define the document(See Sec. 3). An object is stored in the network for each of the fingerprints in a given set. This object stores the GUIDs of all other documents whose set of fingerprints include the fingerprint corresponding to the object. When a new document is to be added to the system, first the document GUID is published. Then each fingerprint is published either by having the GUID of the document appended to the GUID list of the corresponding fingerprint object or creating a new fingerprint object with a list that contains only the GUID of the new document, depending on whether the fingerprint object already existed in the system or not.

Given this, the ATA search for a query document is based on the following: First, the objects corresponding to each of the fingerprints in the fingerprint set is searched within the system. Then the document GUID that appears in more than T lists is used to route messages to that document where T is an adjustable threshold for matching fingerprints.

3 Fingerprints

A fingerprint is fundamentally just a set of characters of a fixed length n. If two documents have the same n characters in a sequence, than they will both share the same fingerprint representing those n characters. More accurately, a single fingerprint is the hash of one of these sets of characters. We hash each fingerprint to a single 128 bit number, and then keep that number to represent the print. This obviously makes more sense than keeping the original character sequence, as it takes less space and is easier to work with.

Essentially, we are interested in finding documents that share many fingerprints. Over all, we would like to be able to detect files which are as little as 25% similar. So, which fingerprints should be compared in the two documents?

An immediate idea is to take every set of n consecutive characters of one file, and compare it to every n character segment of another file. While this approach would provide the detector with the most information, it runs in n^2 time. And for two 10 page papers with 20,000 characters, that's 400,000,000 comparisons! This is obviously too expensive.

Therefore, it is clear that it's not possible to use every fingerprint of two documents to compare them. We need to take a subset of both sets of fingerprints to make the task feasible.

One approach is to take only a segment of the two papers and compare fingerprints from those segments. Then modifications on that segment will significantly reduce any chance of finding similarity, even if the rest of the two documents are identical. Similarly, one could just chose f random fingerprints from each of the documents to compare. But, consider the case where the query document is of length 10,000, and the size of fingerprints is 50. The probability that a particular fingerprint will be selected is 10/10,000, or about 1/1000. Then the probability of the file being matched with itself is 1/1000, a bad probability considering the two documents are identical. Therefore, it is important to find a selection procedure that takes similar kinds of fingerprints from similar documents.

The approach we use here takes every possible n bit fingerprint of a file, and keeps a small subset of these, based on their values. DetectIt choses to keep only those fingerprints which are divisible by 256. This would keep less than .5 percent of the fingerprints in a 10,000 character document[13]. Reducing the number of prints this way helps performance. The algorithm then takes the minimum f of these fingerprints, where f is the number of fingerprints used for a document. If any of those fingerprints happen to appear in other documents, they will be divisible by 256 as well, and it is likely that they will be part of the minimum f of that document's fingerprints. So essentially, we are choosing the same kind of fingerprints from all documents, raising the probability that any given fingerprint is found.

3.1 Fingerprinting Parameters

It is important to point out the many different parameters involved in fingerprinting a document. First there is the "granularity", or the number of characters in a fingerprint. Granularity is the n discussed earlier. Having a very low granularity, say one character in the extreme case, returns many "false positives", or documents that aren't similar, but are marked as such. Almost all documents share very similar one character fingerprints. On the other hand, having a high granularity shifts the problem the other way. Documents probably would not have the same 10,000 character fingerprint unless they were truly identical. It is important to find the appropriate granularity, and that is discussed later in Section 5.1.

Another parameter is the number of fingerprints, or f above. Of course it is useful to have large numbers of fingerprints, but the main trade off here is between accuracy and performance. Having a huge number of fingerprints makes the work to check a single document very difficult. This is because, as mentioned in Section 2.2, every individual fingerprint is turned into a Tapestry query. Large numbers of fingerprints also require much more space in a database. On the other hand, having too few fingerprints helps performance, but makes the reliability of the algorithm decrease.

The last important parameter is the threshold value T, that is the percentage of fingerprints that have to match to claim that two documents are similar. If only a small percentage of fingerprints match, it could be just chance. Therefore, a low threshold might leave a lot of room for false positives. On the other hand, a large threshold might miss many similar documents. Setting T to 90% would mean that 90% of the fingerprints from the documents have to match. Obviously, this would only happen if the documents were extremely similar.

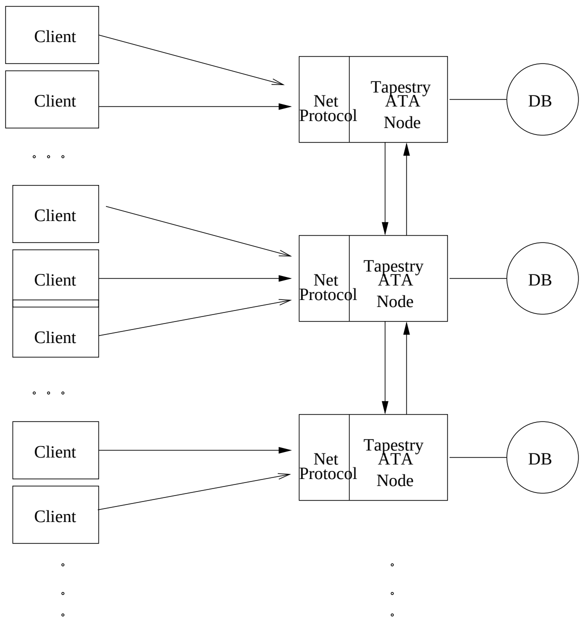

Figure 1: A 3-node DetectIt system

In conclusion, there are trade offs with setting all of these parameters. Finding just the correct values for these parameters is discussed later in Section 5.1.

3.2 Problems with Fixed Size Fingerprints

One way to measure the accuracy of a fingerprinting technique is to compare it to the results from full fingerprinting. While correlation between our technique and the full fingerprinting technique is quite high on documents of similar size, it can become much worse when the set of query documents are of drastically different sizes. If one query is a 1,000 character document, and another one a 100,000 character document, even if the later has the entire 1,000 character document contained in it, the probability of landing a fingerprint in that section can be quite small. It seems that there is a large trade off for the ease of having a fixed size fingerprint. Ideally, more fingerprints for larger documents is desired, but that can prove very difficult to implement in a system.

4 DetectIt

Our system has several components that it operates on. There's the client that takes the documents and the relevant information from the user and sends it over to the net protocol. The net protocol strips the information from the message sent from the client and sends it over to the Tapestry/ATA network. Note that each node on the Tapestry/ATA network has such a net protocol since any node can be used to access the network. The searches in the Tapestry/ATA network are based on information stored in a persistent Database Component on each node. Figure 1 depicts the system components and their interaction with each other. In this section we elaborate on some of the components that need more explanation on their operation.

4.1 The Client Component

A DetectIt client is the user-agent that is used to log on to a Tapestry node in the system. It is written in JAVA using the swing environment and communicates with the Tapestry nodes using simple socket connections.

A user first gives the client a list of the files he/she wants to evaluate, and the authors of these papers. This can be done by adding the papers one by one, or making a script file which adds files and names all at once.

Once DetectIt has the list of papers to evaluate, the user initiates contact with the local Tapestry node and sends it a "check" request, which includes 50 fingerprints, a unique identifier for the paper, the username of the client (usually the professor), the author name, and the date. The Tapestry node then looks for an approximately matching document in the system. If it cannot find one, it returns a message to the user saying the documents cleared. If any documents are suspect (if there exists an approximate match), the node will return the relevant information about the paper that is suspect to the client. Included in this information is the name of professor that originally submitted the paper and a unique identifier for that document. The user can then contact the professor who originally submitted the paper, which can be found using the unique identifier.

With Tapestry's API, the tasks of the client is quite simple. It is in itself just a simple program which sends out the fingerprints and receives the response. The only task of the client in the end is to fingerprint the documents, select the appropriate subset of these fingerprints, and run the communication between the Tapestry/ATA level and the client.

4.2 The Tapestry/ATA Component

Once a Tapestry/ATA node in our system receives the document information from the client side, it performs a search based on the fingerprints as the search algorithm in Section 2.2 suggests. Whether the search returns a suspect document or not, we still publish the document since we want to keep track of all the documents ever submitted. However, since in the ATA layer a document is given a globally unique identifier based on the hash of its fingerprints, storing identical documents with different authors could prove problematic. We solve this problem by using the unique ID received from the client side in addition to the fingerprints hashed all together to derive a globally unique identifier in the system. The publishing procedure is the same as the one explained in Section 2.2.

If the system finds a similar document, then before we send information regarding the suspected paper to the client side, we first compare authors to make sure that the similar paper is not written by the same author, since that doesn't constitute as plagiarism. However, since the user may still be interested in this information, we provide it to the client side so the user can be notified.

4.3 The Database Component

To allow comparisons, one would like to store all papers processed in the system in a database. However, storing full papers in such a database would violate the intellectual properties of authors. Therefore, instead of storing a paper in the system, we store a "digest" of the paper. The digest consists of fingerprints as well as other relevant document information such as author and submission date. We store the digests through the use of two databases. One database keeps track of the mapping between document GUIDs and the set of fingerprints for that document. The other database keeps track of the mapping between a document GUID and the author, the professor, and the date of submission of that document as well as the unique ID provided from the client so that in case of a suspected document the user can locate the paper with ease.

The database is stored on disk for each of the nodes to provide persistent storage. When the system is first started, all objects in the database of the starting node are published so that all other nodes are aware of all digests each node has previously stored.

5 Evaluation

Evaluating copy detection systems can be quite a difficult task. First, a very large data set is needed. Second, we need information on which of those pieces of data were plagiarized from the other. Unfortunately, we could find no such corpus, and so we have to make do with what we had to work with. Our evaluation was based on two corpora. The first is the MUC7 corpus, which is just 200 news articles selected to be about two different topics, air line crashes and missile launches. The second was the bible, chosen for it's large size and easy access.

In this section, we present results for the correctness and performance evaluations of our system. Our correctness evaluation consists of how accurately our system performs in detecting similar documents followed by the performance evaluation that confirms theoretical running time of the fingerprinting scheme. We also experimented with running our system using up to four Tapestry nodes and the results for these experiments are also presented. We conclude our evaluation with a rather big list of shortcomings of our system. All problems noted constitute steps of possible future work.

5.1 Correctness

We conducted many experiments to select parameters for the fingerprinting that would perform well. As stated before, we did not have large enough corpora to do significant studies, but we experimented using the data we did have. First, we looked at setting the appropriate threshold value, T. The threshold value is the percentage of fingerprints that have to match in two documents for them to be considered similar. Figure 2 shows three different threshold values, 10%, 25% and 50%. These experiments were run with a granularity (n) of 100 and 100 fingerprints per document (f). Results show that if we want to find documents that are only 25% similar with these two parameters, the 10% threshold is a good choice. Note that the slope of the middle part of each plot is an indication of how much variability there is in a system. The perfect system would have a graph that looked like a step-graph, where it would be possible to tweak the fall from 100% detection to 0% detection at any percent difference along the x-axis.

We also looked at setting an appropriate fingerprint length, n. We used 10,000 character segments from the bible, and compared them against each other. Since the segments were all unique, we were interested in setting the granularity high enough so that there would be no "false positives", or segments that DetectIt claimed were the same, even though they weren't. Figure 3 shows that we had to make the granularity 30 characters long before eliminating false positives in this extremely small corpus. It is safe to assume that with a lot more data, it's possible to have false positives that would show up, and the fingerprint size would have to increase dramatically. Clearly, the appropriate granularity value is highly dependent on the chosen threshold value. In this experiment, we had a threshold of 10% and we used 50 fingerprints per document.

These experiments, and others like it, led us to set the values that we currently have employed in DetectIt. As implemented, the threshold value is 10%, we take 50 character fingerprints, and we take 100 fingerprints per document. Of course, these numbers are just preliminary values that seem to yield decent results. Further experimentation is needed to set them more appropriatly to the task at hand.

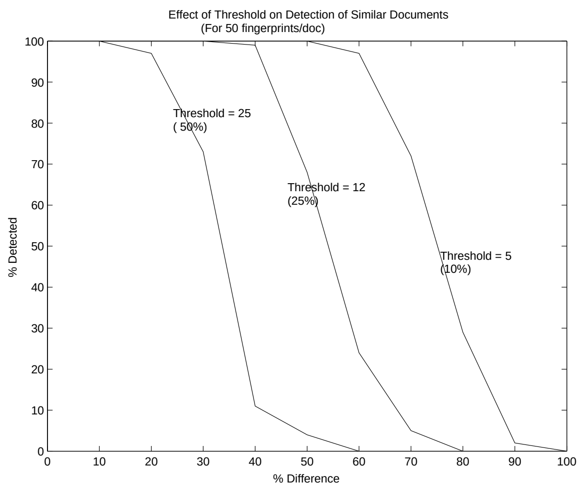

Figure 2: Effect of threshold on detection of similar documents

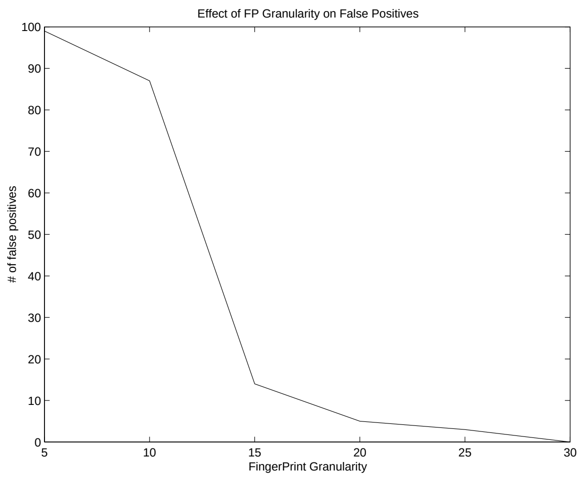

Figure 3: Effect of fingerprint granularity on number of false positives

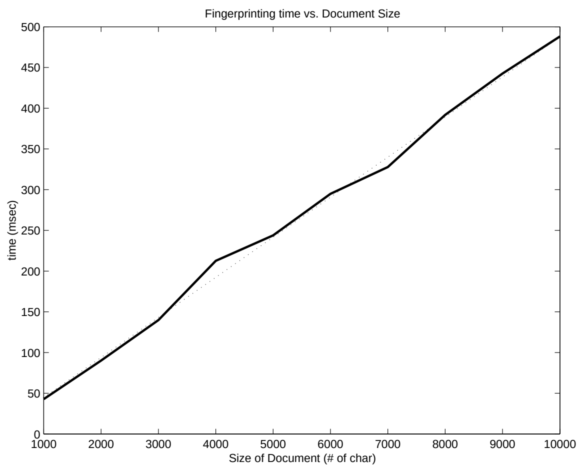

Figure 4: Fingerprinting time vs. document size

5.2 Performance

The performance of the system was definitely suitable for the needs of a user. Figure 4 shows that the fingerprinting time was linear on the size of the document. This makes sense as the algorithm takes every n-character string from a document. The time for fingerprinting a single 10,000 character document was about 500 milliseconds on average.

We also experimented on the time it takes to query a document for different number of nodes in the system. Since we were limited in resources we could test performance on only up to 4 nodes as can be seen in the following table.

	Misses (sec/doc)	Hits (sec/doc)
1 Tapestry Node	.075	.75
2 Tapestry Nodes	.083	.75
3 Tapestry Nodes	.130	.74
4 Tapestry Nodes	.150	.75

The misses column of the table refers to query documents that do not have a similar document in the system and the hits column refers to those that do return a suspected document. As suggested from the numbers, the hits take about constant time as nodes in the system increase whereas an increase can be observed in the time it takes for to return misses. Also clearly for any number of nodes the misses take more time than hits. We believe these results follow those that are suggested by the Tapestry framework.

We conclude the performance evaluation by pointing out that the overall turnaround time for a document of 10,000 characters is less than .75 seconds which we believe to be reasonably fast.

5.3 Shortcomings

Our current version of DetectIt has many shortcomings since it is just a crude version of a copy detection mechanism implemented in a very short period of time. Rather than spending time on optimizations and more functionality, our goal was to bring about the approach of a distributed system and to show that this is a feasible approach. We believe we achieved this goal despite the many shortcomings we'll briefly describe in this section.

One of the problems we have in our system is the "fixed fingerprint size" approach to extracting fingerprints from a document. As we have mentioned in Sec 3.2, for a very large document, a limited number of fingerprints are very unlikely to capture the distinguishing properties of the document. Although we considered supporting variable fingerprints we realized that it would lead to wasted space in the database component of our system if there are few large documents and many short documents.

Another problem we have realized is the fact that our system would not be able to detect a document where each paragraph is taken from different documents. This is a consequence of the fact that our system returns only one - the most similar and closest in network distance - document if a similar version of a queried document was found in the system, due to the design of Tapestry infrastructure.

In literature, different levels of plagiarism have been categorized in the following way: Plagiarized, subset, exact copy, related. A plagiarized document is a document modified slightly with a different author. A subset case refers to the one when one document is exactly contained in another. An exact copy is one where the documents are exactly the same and the author is different. A related document pair would share the same subject therefore using the same terms and maybe voicing the same opinions. A desirable copy detection mechanism would be able to distinguish between these categories. Our system at this point cannot. We only return "suspected" or "clear" as a result since we only check for matching number of fingerprints greater than a fixed threshold, which is currently 5, or 10%.

Many of the fee-based services offered on the web test query documents against other documents available on the web. For our system to work against Internet-plagiarism, somebody would have to enter all such documents into our system. It could be useful to implement some sort of an update mechanism that would search such documents on the web periodically and add them to our system automatically.

Another improvement to our system would be to add support for different formats of documents. Currently we only support text documents, but we think it would be worth the effort to support other documents such as PDF and Postscript. However, supporting such documents is not as trivial as it sounds due to the amount of noise they introduce into the documents during the conversion. For example, two mathematical documents with numerous embedded equations would lose all such equations during the conversion process. Then it would be rather difficult to detect the similarity between two such mathematical documents.

6 Related Work

The systems aspect of our work closely follows that of Zhou, et al.[15]. Their work consisted of creating the Approximate Text Addressing layer on Tapestry and implementing an application called SpamWatch, used to detect spam mail. Our work is different from theirs in the following ways: firstly, our application has different storage requirements regarding the paper related information on the tapestry nodes. We store much more information about a document so that when plagiarism is suspected, it is easy for the user to trace down a given document, whereas in SpamWatch no such storage is necessary. Secondly, our handling of suspected documents is slightly different than theirs: we store the information related to every single document processed, since our goal is to have a big database that a user can test against. Theirs involved only storage of documents that were identified as spam. And finally, our user interface differs greatly from theirs. DetectIt was implemented as an independent application with a necessary GUI, whereas SpamWatch was an add-in to Microsoft Outlook.

From the copy detection point of view, one can find many papers in literature describing different approaches to the issue of plagiarism. Our approach fits in the category of "registry servers", where a query document is checked against a repository of original documents. There has been numerous implementations of copy detection systems in this category such as SCAM [9], [10], COPS [1], CHECK [11], Koala [5], and SE(Signature Extraction) [3]. The main difference between our system and these systems is that our system is a peer-to-peer decentralized system whereas in these systems all clients connect to one server that constitutes a single point of failure. The systems differ among each other based on algorithms used to detect similarities, the accuracy of detected similarities and enhancements to decrease space and time requirements. Notably, most systems use the fingerprinting approach by Manber [13] or a variation of it. Others use a method called "ranking" which was borrowed from the field of Information Retrieval [6].

To our knowledge there is only one *fairly* distributed approach to the problem of copy detection as described in [4]. This is a distributed version of SCAM, thus the name "dSCAM". dSCAM can go through multiple databases to search for a query document, unlike the other one-server systems. Given a large list of databases with some meta-data information, the system first flags the databases that contains at least one similar document to the one that's queried. Then these databases are searched using a slightly different algorithm to minimize the number of documents to check against and other such querying costs. dSCAM is quite different in its approach and methods compared to DetectIt in the following ways: first of all, with the use of the transparency of Tapestry, we see many databases unified as one; there's no need to keep track of database meta-data. The search is conducted based on the routing algorithms of Tapestry which avoid having to check every other document in the whole system. Also, our system supports a peer-to-peer design whereas dSCAM doesn't. A feature in dSCAM that is not possible in our approach is that it doesn't require the participating databases to be a part of the system. It provides methods to be able to search and retrieve documents in disconnected databases.

7 Conclusions

We presented DetectIt, a peer-to-peer plagiarism detection system built on the Tapestry framework. Therefore our system inherits Tapestry's scalability, fault tolerance and ease of maintenance. We chose to use such a distributed approach due to the nature of the problem, where effectiveness increases with the number of users. Our method for detecting similar documents relies on the concept of fingerprinting.

The accuracy of a fingerprinting technique is highly dependent on the choice of parameters. Our goal was to prove that a distributed approach to plagiarism detection could be successful. Therefore, we did not perform a thorough analysis to optimize the parameters. Our chosen parameters allow detection of similarity between documents that are up to 70% different. However, changing these parameters in the system is a trivial task.

Our results show that this approach to the problem is quite feasible. Our overall turn around time to query a document was less that .75 seconds, suitable for regular use of the program. The platform independent client allows widespread deployment of the system with very little maintenance requirements.

References

[1] S. Brin, J. Davis, and H. Garcia-Molina "Copy Detection Mechanisms for Digital Documents" In Proc. of ACM SIGMOD Ann. Conf., San Francisco, CA, 1995

[2] J. Weinstein and C. Dobkin "Plagiarism in U.S. Higher Education: Estimating Internet Plagiarism Rates and Testing a Means of Deterrence" University of California, Berkeley, Nov. 2002,

[3] R.A. Finkel, A. Zaslavsky, K. Monostori, H. Schmidt "Signature Extraction for Overlap Detection in Documents" Twenty-Fifth Australasian Computer Science Conference (ACSC2002)

[4] H. Garcia-Molina,L. Gravano and N. Shivakumar "dSCAM: Finding Document Copies Across Multiple Databases (1996)" Proc. of 4th Intl. Conf. on Parallel and Distributed Information Systems

[5] N. Heintze "Scalable Document Fingerprinting" 1996 USENIX Workshop on Electronic Commerce

[6] T.C. Hoad and J. Zobel "Methods for Identifying Versioned and Plagiarized Documents" Jour.of the American Soc. for Information Science and Technology. (To appear)

[7] S. Ratnasamy, P. Francis, M. Handley, R. Karp and S. Shenker "Scalable Content Addressable Network" Proc. of ACM SIGCOMM 2001

[8] A. Rowston and P. Druschel "Pastry:Scalable, decentralized object location and routing for large-scale peer-to-peer systems" Proc. 18th IFIP/ACM Intl. Conf. on Distributed Systems Platforms, Nov. 2001

[9] N. Shivakumar and H. Garcia-Molina "Building a Scalable and Accurate Copy Detection Mechanism" Proc. of 1st Intl. Conference on Digital Libraries, 1996

[10] N. Shivakumar and H. Garcia-Molina "SCAM: A Copy Detection Mechanism for Digital Documents" Proc. of the 2nd Ann. Conf. on the Theory and Practice of Digital Libraries

[11] A. Si, H.V. Leong and R.W.H. Lau "CHECK: A Document Plagiarism Detection System" In Proc.of ACM Symp. for Applied Computing, pp70-77, Feb 1997

[12] I. Stoica, R. Morris, D. Karger, M.F. Kaashoek, H. Balakrishnan "Chord:Scalable Peer-to-Peer Lookup Service for Internet Applications" SIGCOMM'01 Aug 27-31, 2001, San Diego, California

[13] U. Manber "Finding similar files in a large file system." In Proceedings of Winter USENIX Conference, 1994

[14] B. Zhao, A.D. Joseph and J. Kubiatowicz "Tapestry: An Infrastructure for Fault-tolerant Wide-area Location and Routing" Tech. Rep. UCB/CSD-01-1141, U.C.Berkeley, 2001

[15] F. Zhou, L. Zhouang, B. Zhao, L. Huang, A.D. Joseph and J. Kubiatowicz "Approximate Object Location and Spam Filtering and Peer-to-peer Systems" Proc. of ACM/IFIP/USENIX Intl. Middleware Conf. , 2003