Appears in *Proc. of ACM/IFIP/USENIX Intl. Middleware Conf.* (Middleware 2003)

Approximate Object Location and Spam Filtering on Peer-to-peer Systems

Feng Zhou, Li Zhuang, Ben Y. Zhao, Ling Huang,
Anthony D. Joseph, and John Kubiatowicz

Computer Science Division, U. C. Berkeley
`{zf, zl, ravenben, hling, adj, kubitron}@cs.berkeley.edu`

Abstract. Recent work in P2P overlay networks allow for decentralized object location and routing (DOLR) across networks based on unique IDs. In this paper, we propose an extension to DOLR systems to publish objects using generic *feature vectors* instead of content-hashed GUIDs, which enables the systems to locate similar objects. We discuss the design of a distributed text similarity engine, named *Approximate Text Addressing (ATA)*, built on top of this extension that locates objects by their text descriptions. We then outline the design and implementation of a motivating application on ATA, a decentralized spam-filtering service. We evaluate this system with 30,000 real spam email messages and 10,000 non-spam messages, and find a spam identification ratio of over 97% with zero false positives.

Keywords: Peer-to-peer, DOLR, Tapestry, spam filtering, approximate text matching

1 Introduction

Recent work on structured P2P overlay networks ([5, 18], [15], [11], [10]) utilize scalable routing tables to map unique identifiers to network locations, providing interfaces such as Decentralized Object Location and Routing (DOLR) and Distributed Hashtables (DHT). They allow network applications such as distributed file systems and distributed web caches to efficiently locate and manage object replicas across a wide-area network.

While these systems excel at locating objects and object replicas, they rely on known Globally Unique IDentifiers (GUID) for each object, commonly generated by applying a secure hash function to the object content. This provides a highly specific naming scheme, however, and does not lend itself to object location and management based on semantic features.

To address this problem, we propose an approximate location extension to DOLR systems to publish and locate objects using generic *feature vectors* composed of a number of values generated from its description or content. Any object can be addressed by a feature vector matching a minimal threshold number of entries with its original feature vector. Based on this extension, we propose an Approximate Text Addressing (ATA) facility, which instantiates the approximate location extension by using block text fingerprints as features to find matches between highly similar text documents. To validate the ATA design as well as the approximate object location extension, we design a decentralized spam-filtering application that leverages ATA to accurately identify

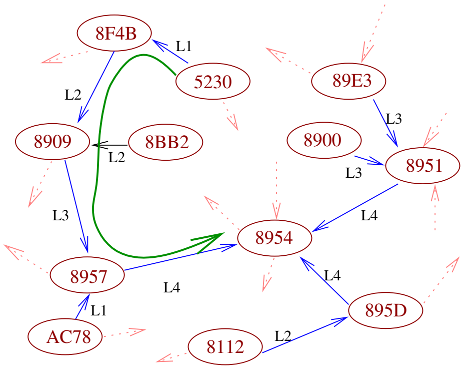

Fig. 1. *Tapestry routing example.* Path taken by a message from node `5230` for node `8954` in Tapestry using hexadecimal digits of length 4 (65536 nodes in namespace).

junk email messages despite formatting differences and evasion efforts by spammers. We evaluate the accuracy of our fingerprint vector scheme via simulation and analysis on real email data, and explore the trade-offs between resource consumption and search accuracy.

The rest of this paper is as follows: Section 2 briefly describes existing work in P2P overlays. Section 3 presents our approximation extension to DOLR systems and a prototype implementation. Section 4 describes the design of ATA and Section 5 discusses the design of the decentralized spam filter. Section 6 presents simulation and experimental results, followed by a discussion of related work in Section 7 and status and future work in Section 8. Finally, we provide a mathematical analysis of the robustness of text-based fingerprinting in Appendix A.

2 Background: Structured P2P Overlays

In this section, we first present background material on structured P2P overlays. Different protocols differ in semantics details and performance objectives. While we present our work in the context of Tapestry for performance reasons, our design is general, and our results can be generalized to most structured P2P protocols.

2.1 Routing

Tapestry is an overlay location and routing layer first presented in [18], with a rigorous treatment of dynamic algorithms presented in [5]. Like other structured P2P protocols, object and node IDs are pseudo-randomly chosen from the namespace of fixed-length bit sequences with a common base (e.g. Hex). Tapestry uses local routing tables at each node to route messages incrementally to the destination ID digit by digit (e.g., `4***` $\Longrightarrow$ `45**` $\Longrightarrow$ `459*` $\Longrightarrow$ `4598` where `*`'s represent wildcards). A node N has a neighbor map with multiple levels, where each level represents a matching prefix up to a digit position in the ID. Each level of the neighbor map contains a number of entries

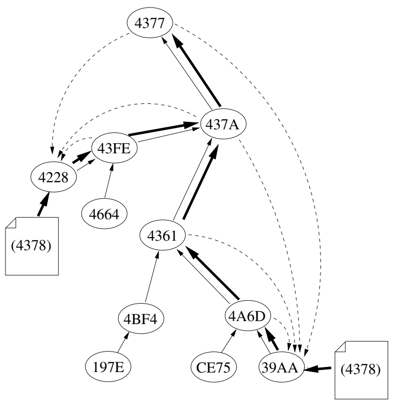

Fig. 2. *Publication in Tapestry.* To publish object `4378`, server `39AA` sends publication request towards root, leaving a pointer at each hop. Server `4228` publishes its replica similarly. Since no `4378` node exists, object `4378` is rooted at node `4377`.

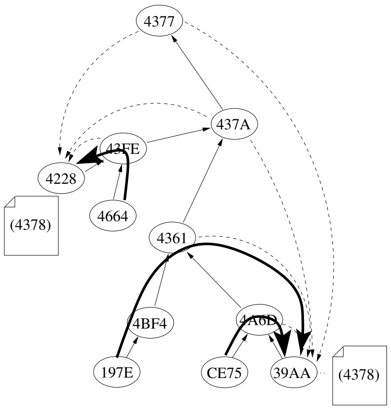

Fig. 3. *Object Location in Tapestry:* Three different location requests. For instance, to locate GUID `4378`, query source `197E` routes towards the root, checking for a pointer at each step. At node `4361`, it encounters a pointer to server `39AA`.

equal to the base of the ID, where the i^{th} entry in the j^{th} level is the location of the node *closest in network latency* that begins with *prefix*$_{j-1}(N) + i$.

To forward on a message from its n^{th} hop router, Tapestry examines its $n+1^{th}$ level routing table and forwards the message to the link corresponding to the $n+1^{th}$ digit in the destination ID. This routing substrate provides efficient location-independent routing within a logarithmic number of hops and using compact routing tables. Figure 1 shows a Tapestry routing mesh.

2.2 Data Location

In Tapestry, a server S makes a local object O available to others by routing a "publish" message to the object's "root node," the live node O's identifier maps to. At each hop along the path, a location mapping from O to S is stored. Mappings for multiple replicas are stored sorted according to distance from the local node. See Figure 2 for an example of object publication. Here two replicas of the same object are published. A client routes a query message towards the root node. The message queries each hop router along the way, and routes towards S when it finds the O to S location mapping. Note that for nearby objects, query messages quickly intersect the path taken by publish messages, resulting in quick search results that exploit locality [18]. See Figure 3 for an example of object location. Notice how locality is exploited by directing location requests to nearby replicas.

3 Approximate DOLR

DOLR systems like Tapestry provide deterministic, scalable, and efficient location and routing services, making them attractive platforms for deploying wide-area network

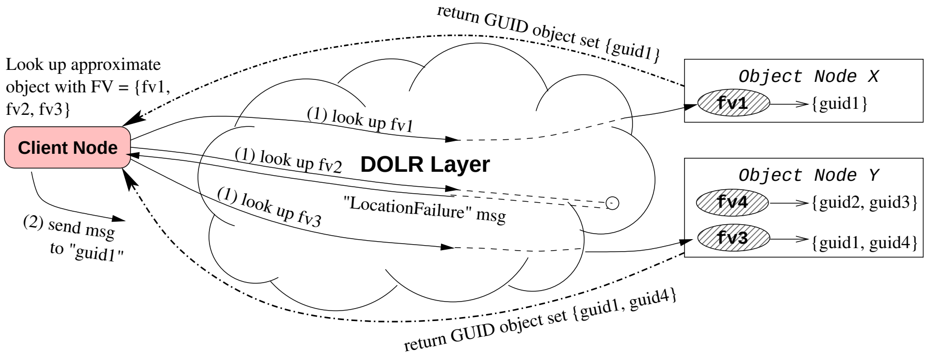

Fig. 4. *Location of an approximate object.* Client node wants to send a message to all objects with at least 2 feature in {`fv1, fv2, fv3`}. It first sends lookup message to feature `fv1`, `fv2` and `fv3`. `fv2` does not exists. A `Location Failure` message is sent back. `fv1` is managed by object node `X`. It sends back a list of IDs of all objects having feature `fv1`, which is {`guid1`}. Similar operation is done for feature `fv3`, whose IDs list {`guid1, guid4`}. Client node counts the occurrence of all IDs in all lists and finds out `guid1` to be the ID it is looking for. It then sends the payload message to object `guid1` using Tapestry location message.

applications. Files, in particular, can be located efficiently if their canonical name is known. Previous approaches, however, generate Globally Unique IDentifiers (GUID) by a secure hash (e.g. SHA-1) of the content. This approach significantly limits the usability of the system in scenarios where users do not known exact names of objects, but rather perform searches based on general characteristics of the system. In particular, these scenarios might include searches for data that closely approximates, or is similar to known data with certain properties. Examples might include searching for audio or video that matches existing works in content features, or searching or lightly modified replicas of existing data.

3.1 Approximate DOLR Design

Here we propose an extension to DOLR, *Approximate DOLR*, as a generic framework to address some of the needs of these applications. In an ADOLR system, we apply application-specific analysis to given objects to generate *feature vectors* that describe its distinctive features, and provide a translation mechanism between these application-driven features and a traditional GUID obtained from a secure content hash of the object contents.

This query ability on features applies to a variety of contexts. In the world of multimedia search and retrieval, we can extract application-specific characteristics, and hash those values to generate feature vectors. Any combination of field to value mappings can be mapped to a feature vector, given a canonical ordering of those fields. For example, this can be applied to searching for printer drivers given printer features such as location, manufacturer, and speed. If features are canonically ordered as `[location,` `manufacturer, speed]`, then an example feature vector might be `[hash(443 Soda), hash(HP), hash(12ppm)]`.

Each member of the vector, a *feature*, is an application-specific feature encoded as a hashed identifier. For each feature *f*, an object (*feature object*) is stored within the network. The feature object is a simple object that stores the list of GUIDs of all objects whose feature vectors include *f*. Clients searching for objects with a given feature set finds a set of feature objects in the network, each associated with a single feature, and selects the GUIDs which appear in at least *T* feature objects, where T is a tunable threshold parameter used to avoid false positives while maintaining the desired generality of matches.

The "publication" of an object O in an ADOLR system proceeds as follows. First, its content-hash derived GUID is first published using the underlying P2P DOLR layer. This assures that any client can route messages to the object given its GUID. Next, we generate a feature vector for O. For each feature in the vector, we try to locate its associated feature object. If such an object is already available in the system, we append the current GUID to that object. Otherwise, we create a new feature object identified by the feature, and announce its availability into the overlay.

To locate an object in an ADOLR system, we first retrieve the feature object associated with each entry of the feature vector. We count the number of distinct feature objects each unique GUID appears in, and select the GUID(s) that appear in a number greater than some preset threshold. The GUID(s) are then used to route messages to the desired object.

The ADOLR API is as follows:

- **PublishApproxObject (FV, GUID)**. This publishes the mapping between the **feature vector** and the GUID in the system. A feature vector is a set of feature values of the object, whose definition is application specific. Later, one can use the feature vector instead of the GUID to search for the object. Notice that **PublishApproxObject** only publishes the mapping from `FV` to `GUID`. It does not publish the object itself, which should be done already using publish primitive of Tapestry when **PublishApproxObject** is called.
- **UnpublishApproxObject (FV, GUID)**. This removes the mapping from the `FV` to the `GUID` if this mapping exists in the network, which is the reverse of **PublishApproxObject**.
- **RouteToApproxObject (FV, THRES, MSG)**. This primitive routes a message to the location of all objects which overlap with our queried feature vector `FV` on more than `THRES` entries. The basic operations involve for each feature, retrieving a list of GUIDs that share that feature, doing a frequency count to filter out GUIDs that match at least `THRES` of those features, and finally routing the payload message `MSG` to them. For each object in the system with feature vector FV^*, the selection criterion is:

  $$|FV^* \bigcap FV| \geq THRES \;\; AND \;\; 0 < THRES \leq |FV|$$

  The location operation is deterministic, which means all existing object IDs matching the criterion will be located and be sent the payload message. However, it is important to notice that this does not mean every matching object in the system will receive the message, because each object ID may correspond to multiple replicas, depending on the underlying DOLR system. The message will be sent to one replica of each matching object ID, hopefully a nearby replica if the DOLR utilizes locality.

With this interface, we reduce the problem of locating approximate objects on P2P systems to finding a mapping from objects and search criteria to feature vectors. The mapping should maintain similarity relationships, such that similar objects are mapped to feature vectors sharing some common entries. We show one example of such a mapping for text documents in Section 4.

3.2 A Basic ADOLR Prototype on Tapestry

Here we describe an Approximate DOLR prototype that we have implemented on top of the Tapestry API. The prototype serves as a proof of concept, and is optimized for simplicity. The prototype also allows us to gain experience into possible optimizations for performance, robustness and functionality.

The prototype leverages the DOLR interface for publishing and locating objects, given an associated identifier. When **PublishApproxObject** is called on an object O, it begins by publishing O's content-hashed object GUID using Tapestry. Then the client node uses Tapestry to send messages to all feature objects involved. Tapestry routes these messages to the nodes where these feature objects are stored. These nodes then add the new object GUID to the list of GUIDs inside the feature object. If any feature object is not found in the network, the client node receives a `LocationFailure` message, creates a new feature object containing the new object, and publishes it.

For the **RouteToApproxObject** call, the client node first uses Tapestry location to send messages to all feature objects, asking for a list of IDs associated with each feature value. Nodes where these feature objects reside receive these messages, do the lookup in their maps and send back the result. **LocationFailure** messages are sent back for nonexistent feature objects, and are counted as an empty ID list. The client node counts the occurrence of each GUID in the resulting lists. GUIDs with less than the threshold number of counts are removed. Finally, the message in this call is sent to the remaining object GUIDs An example of executing a **RouteToApproxObject** call is shown in Figure 4.

Note that an analogous system can be implemented on top of a distributed hash table (DHT) abstraction on P2P systems. Instead of routing messages to previously published feature objects, one would retrieve each feature object by doing a **get** operation, appending the new GUID, and putting the object back using **put**.

3.3 Optimizing ADOLR Location

Our initial description of the **RouteToApproxObject** operation involves several round-trips from the client node to nodes where the feature objects are stored. We propose two optimizations here that eliminates a network round-trip, reducing overall latency to that of a normal **RouteToObject** in a DOLR system at the cost of keeping a small amount of state on overlay nodes. The first optimization involves a client node caching the result

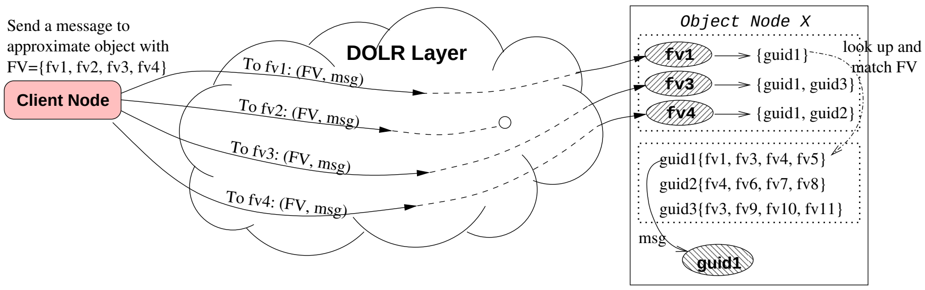

Fig. 5. *Optimized ADOLR location.* Client node wants to route a message to a feature vector {`fv1, fv2, fv3, fv4`}. It sends message to each identifier `fv1`, `fv2`, `fv3`, `fv4`. `fv2` doesn't exist, so no object node receives this message. When object node `X` receives the messages to `fv1`, `fv3` and `fv4`, it scans its local storage for all IDs matching `fv1`, `fv3` and `fv4`, which is `guid1`. Then, object node `X` sends `msg` to `guid1`.

of translating a feature vector to a GUID. now all future messages to the same feature vector are routing to the cached GUID.

The second optimization is more complex, and illustrated in Figure 5. Normally, the client node retrieves a set of feature objects, counts GUID occurrences locally, then routes a message to the resulting GUID(s). The intuition here is that if features are identified as hashed keys with reasonably low collision rates, each feature will likely only identify a small number (one or two) of objects with that feature. Furthermore, multiple feature objects are likely to be colocated together along with the object they identify, because new feature objects are created by the same node where the object is stored. Another way to look at this is that the feature object is in most cases published at the same time with the object itself by the same node. This implies we can route the application-level message to each feature in the feature vector, and expect it to arrive at the node where the desired object is stored.

The key change here is that any node that is storing a feature object, (a file providing a mapping from a feature to all GUIDs that share that feature), also stores the feature vectors of each of those GUIDs. Routing a message to a feature vector $\{X, Y, Z\}$ means sending the message to each identifier X, Y, and Z. Each message also includes the entire feature vector we're querying for. When a node receives such a message, it immediately scans its local storage for all feature objects matching X, Y, or Z. For each GUID in these feature objects, the node determines the amount of overlap between its feature vector and the queried feature vector. If the overlap satifies the query threshold, the message is delivered to that GUID's location.

This implies that any of the query messages contains enough information for a node to completely evaluate the ADOLR search on local information. If any locally stored feature objects contain references to matching objects, they can be evaluated immediately to determine if it satisfies the query. Because each message contains all necessary information to deliver the payload to the desired GUID, the set of messages sent to X, Y, and Z provide a level of fault-resilience against message loss. Finally, the determination of the desired GUID can occur when the first message is received, instead of waiting for all messages to arrive.

The translation from the feature vector to one or more GUIDs occurs in the network, not the client node. This provides significant communication savings.

Nodes need to keep more state to support this optimization, however. In addition to storing feature objects (that keep the mapping between feature values and GUIDs), they also need to keep track of previously resolved feature vectors in order to drop additional requests for the same feature vector. This state can be stored on a temporary basis, and removed after a reasonable period of time (during which any other requests for the same feature vector should have arrived).

3.4 Concurrent Publication

There is one problem with the **PublishApproxObject** implementation described above. The lookup of feature objects and publication of new feature objects are not atomic. This can result in multiple feature objects for the same feature value being published if more than one node tries to publish an object with this feature value concurrently.

We propose two solutions. First, we can exploit the fact that every object is mapped to a unique root node and serialize the publication on the root node. Every node is required to send a message to the root node of the feature value to obtain a leased lock before publishing the feature object. After the lock is acquired by the first node, other nodes trying to obtain it will fail, restart the whole process, and find the newly published feature object. This incurs another round-trip communication to the root node.

In a more efficient "optimistic" way to solve this problem, the client node always assumes the feature object does not exist in the network. It tries to publish the object without doing a lookup beforehand. When the publication message travels through the network, each node checks whether it knows about an already published feature object with the same feature value. If such an object does exist, some node or at least the root will know about this. The node who detects this then cancels this publication and sends an message to the existing feature object to "merge" the new information. This process is potentially more efficient since conflicts should be rare. In general, the operation is accomplished with a single one-way publication message.

This optimistic approach can easily be implemented on top of DOLRs such as Tapestry using the recently proposed common upcall interface for peer to peer (P2P) overlays [2]. This proposed upcall interface allows P2P applications to override local routing decisions. Specifically, a node can "intercept" the publication message and handle conflicts as specified above.

4 Approximate Text Addressing

In this section, we present the design for the Approximate Text Addressing facility built on the Approximate DOLR extension, and discuss design decisions for exploring trade-offs between computational and bandwidth overhead and accuracy.

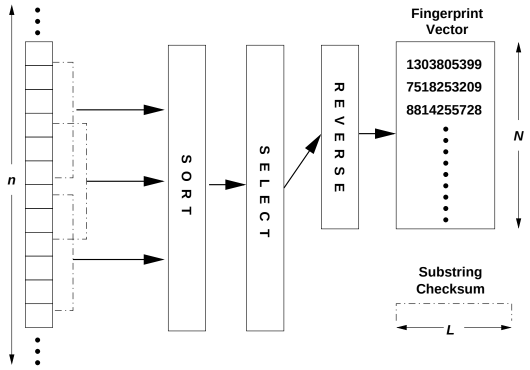

Fig. 6. *Fingerprint Vector.* A fingerprint vector is generated from the set of checksums of all substrings of length L, post-processed with sort, selection and reverse operations.

4.1 Finding Text Similarity

Our goal is to efficiently match documents distributed throughout the network that share strong similarities in their content. We focus here on highly similar files, such as modified email messages, edited documents, or news article published on different web sites.

The algorithm is as follows. Given a text document, we use a variant of block text fingerprinting first introduced in [7] to generate a set of fingerprints. The fingerprint vector of a document is used as its feature vector in publication and location, using the Approximate DOLR layer.

To calculate a block text fingerprint vector of size N for a text document, we divide the document into all possible consecutive substrings of length L. A document of length n characters will have $(n - L + 1)$ such strings. Calculating checksums of all such substrings is a fast operation which scales with n. We sort the set of all checksums by value, select a size N subset with the highest values, and reverse each checksum by digit (i.e. $123 \Rightarrow 321$). This deterministically selects a random set without biasing the ID for prefix or numerical routing.

L is a parameterized constant chosen for each application to tune the granularity of similarity matches. For example, a size L of 50 might work well for email, where complete sentences might account for one substring; but less well for source code, where code fragments are often much longer in length. Figure 6 illustrates the fingerprint process. The calculation is not expensive. Our Java prototype has a processing throughput of $> 13MB/s$ for $L = 50$ on a 1Ghz PIII laptop.

4.2 Trade-offs

There are obvious trade-offs between network bandwidth used and the accuracy of the search. First, the greater the number of entries N in a vector, the more accurate the match (less false-positives), and also the greater number of parallel lookup requests for each document. Next, the distance each lookup requests travels directly impacts bandwidth consumption on the overall network. ATA-enabled applications [1] can benefit from exploiting network-locality by matching against similar documents nearby in the network via a DOLR/DHT with object location locality such as Tapestry. Finally, a trade-off exists between the number of publishers (those who indicate they have a particular document), and the resources required for a client to find a match in their query. Bandwidth and accuracy can be tuned by placing a Time-to-Live (TTL) field on the lookup query, constraining the scope of query messages. Clients who fail to find a match may publish their own documents, improving lookup performance for other clients. These are explored in detail in Section 6.

5 Decentralized Spam Filtering

Spam, or unsolicited email, wastes time and valuable network resources, causing headaches for network administrators and home users alike. Currently the most widely-deployed spam filtering systems scale to a university- or company- wide network, and use keyword matching or source address matching [13]. Although easy to deploy and manage, these systems often walk a fine line between letting spam through and blocking legitimate emails. Our observation is that human recognition is the only fool-proof spam identification tool. Therefore, we propose a decentralized spam filter that pools the collective spam recognition results of all readers across a network.

There already exist centralized collaborative spam filtering systems, such as SpamNet [14], which claims to be peer-to-peer but actually uses a Napster-like architecture. To our knowledge ours is the first attempt to build a truly decentralized collaborative spam filtering system. Compared to alternative university-wide centralized collaborated designs, the most important benefit of our wide-area decentralized design lies in the fact that the effectiveness of the system grows with the number of its users. In such a system with huge number of users world-wide, it is highly probable that every spam email you receive has been received and identified by somebody else before because of the large number of users. The deterministic behavior of DOLR systems will prove useful, because when any single peer publishes information about a specific email, that piece of information can be deterministically found by all clients. Therefore we can expect this system to be more responsive to new spam than systems in which different nodes publish/exchange spam information at certain intervals, such as [3]. Additionally, decentralized systems provide higher availability and resilience to failures and attacks than similar centralized solutions such as SpamNet.

5.1 Basic Operation

The decentralized spam filtering system consists of two kinds of nodes, user agents and peers. User agents are extended email client programs that users use. They query peers when new emails are received and also send user's feedback regarding whether a certain email is or is not spam to peers. A peer is a piece of long-running software that

---

[1] Some example applications include spam filters, plagiarism detection and news article clustering.

is installed typically on a university, department or company server that speaks to other peers worldwide and forms a global P2P network.

When an email client receives a message from the server, the user agent extracts the body of the mail, drops format artifacts like extra spaces and HTML tags, generates a fingerprint vector, and sends it to a peer in the DOLR system. The peer in turn queries the network using the Approximate DOLR API to see if information on the email has been published. If a match is found, and it indicates the email is spam, the email will be filed separately or discarded depending on user preference. Otherwise, the message is delivered normally. If the user marks a new message as spam, the user agent *marks* the document, and tells the peer to publish this information into the network.

5.2 Enhancements and Optimizations

The basic design above allows human identification of spam to quickly propagate across the network, which allows all users of the system to benefit from the feedback of a few. There are several design choices and optimizations which will augment functionality and reduce resource consumption.

Our fingerprint vectors make reverse engineering and blocking of unknown emails very difficult. With the basic system, however, attackers can block well known messages (such as those from group mailing lists). We propose to add a voting scheme on top of the publish/search model. A count of positive and negative votes is kept by the system, and each user can set a threshold value for discarding or filing spam using the count as a confidence measure. A central authority controls the assignment and authentication of user identities. A user agent is required to authenticate itself before being able to vote for or against an email. Thus we can restrict the number of votes a certain user agent can perform on a certain email.

Another type of attack is for spammers to find arbitrary text segments with checksum values more likely to be selected by the fingerprint selection algorithm. By appending such “preferred” segments to their spam emails, spammers can fix the resulting email fingerprint vectors to attempt to avoid detection. Note that this attack can only succeed if a continuous stream of unique text segments are generated and an unique segment is appended to each spam message. This places a significant computational overhead on the spammer that scales with the number of spam messages sent. Additionally, mail clients can choose randomly from a small set of fingerprint calculation algorithms. Different fingerprinting methods can include transforming the text before calculating the checksums, changing the checksum method, or changing the fingerprint selection method. To circumvent this, the spammer would need to first determine the set of fingerprint algorithms, and then append a set of preferred segments, each segment overcoming a known selection algorithm. While different fingerprint algorithms generate distinct spam signatures for the same spam, partitioning the user population and reducing the likelihood of a match, it also requires significantly more computational overhead to overcome.

Optimizations can be made for centralized mail servers to compute fingerprint vectors for all incoming messages. These vectors can be compared locally to identify “popular” messages, and lookups performed to determine if they are spam. Additionally, the server can attach precomputed fingerprint vectors and/or spam filtering results as custom headers to messages, reducing local computation, especially for thin mail clients such as PDAs.

6 Evaluation

In this section, we use a combination of analysis, experimentation on random documents and real emails to validate the effectiveness of our design. We look at two aspects of fingerprinting, robustness to changes in content and false positive rates. We also evaluate fingerprint routing constrained with time-to-live (TTL) fields, tuning the trade-off between accuracy and network bandwidth consumption.

6.1 Fingerprint on Random Text

We begin our evaluation by examining the properties of text fingerprinting on randomly generated text. In particular, we examine the effectiveness of fingerprinting at matching text after small modifications to their originals, and the likelihood of matching unrelated documents (false positive rate).

Robustness to Changes in Content We begin by examining the robustness of the fingerprint vector scheme against small changes in a document, by measuring the probability a fingerprint vector stays constant when we modify small portions of the document. We fix the fingerprint vector size, and want to measure the robustness against small changes under different threshold constants (THRES).

In experiments, we take 2 sets of random text documents of size 1KB and 5KB, which match small- and large-sized spam messages respectively, and calculate their fingerprint vectors before and after modifying 10 consecutive bytes. This is similar to text replacement or mail merge schemes often used to generate differentiated spam. We measure the probability of at least $THRES$ out of $|FV|$ fingerprints matching after modification as a function of threshold ($THRES$) and the size of the document (1KB or 5KB). Here, fingerprint vector size is 10, $|FV| = 10$. We repeat that experiment with a modification of 50 consecutive bytes, simulating the replacement of phrases or sentences and finally modifying 5 randomly placed words each 5 characters long.

In addition to the simulated experiments, we also developed a simple analytical model for these changes based on basic combinatorics. We present this model in detail in Appendix A. For each experiment, we plot analytical results predicted by our model in addition to the experimental results.

In Figure 7, we show for each scenario experimental results gathered on randomized text files, by comparing fingerprint vectors before and after modifications. From Figure 7, we can see the model in Appendix A predicts our simulation data almost exactly under all three patterns of modification. More specifically, modifying 10 characters in the text only impacts 1 or 2 fingerprints out of 10 with a small probability. This means setting any matching threshold below 8 will guarantee near 100% matching rate. When we increase the length of the change to 50 characters, the results do not change significantly, and still guarantee near perfect matching with thresholds below 7. Finally,

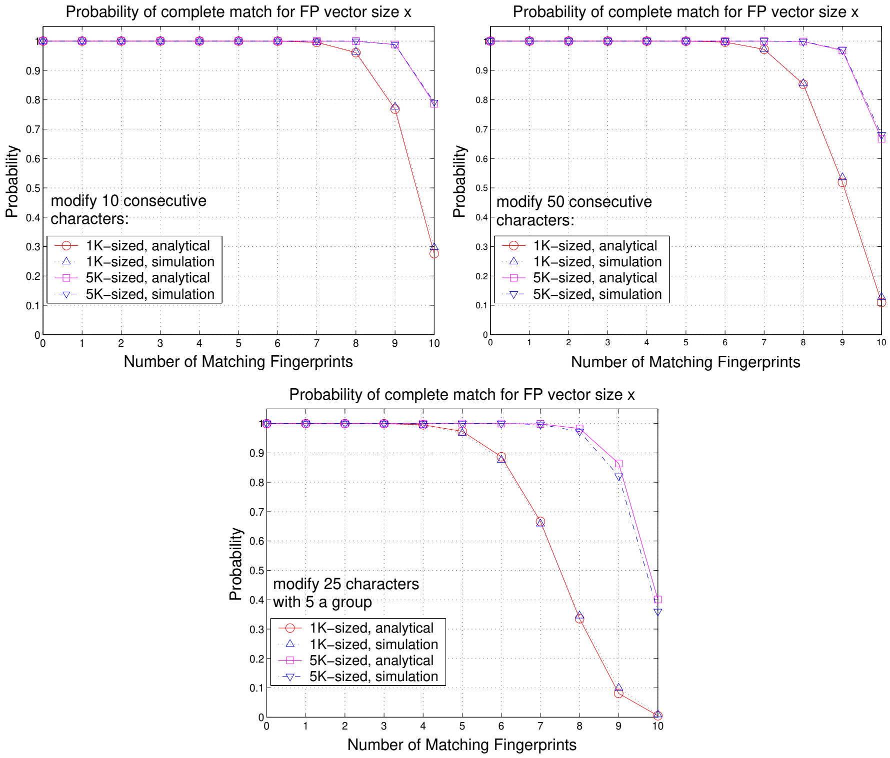

Fig. 7. *Robustness Test (Experimental and Analytical).* The probability of correctly recognizing a document after modification, as a function of threshold. $|FV| = 10$.

we note that multiple small changes (in the third experiment) have the most impact on changing fingerprint vectors. Even in this case, setting a threshold value around 5 or less provides a near perfect matching rate.

Avoiding False Positives In addition to being robust under modifications, we also want fingerprint vectors to provide a low rate of false positives (where unrelated documents generate matching entries in their vectors). In this section, we evaluate fingerprint vectors against this metric with simulation on random text documents. In Section 6.2, we present similar tests on real email messages.

First, we generate 100,000 random text files and find document pairs that match 1 out of 10 fingerprint entries. This experiment is done for different file sizes ranging from 1KB to 64KB. Figure 8 shows the resulting false positive rate versus the file size. While the results for one fingerprint match are already low, they can be made statistically insignificant by increasing the fingerprint matches threshold ($THRESH$) for a "document match." Out of all our tests (5×10^9 pairs for each file size), less than 25 pairs of files (file size > 32K) matched 2 fingerprints, no pairs of files matched more than 2 fingerprints. This result, combined with the robustness result, tells us that on

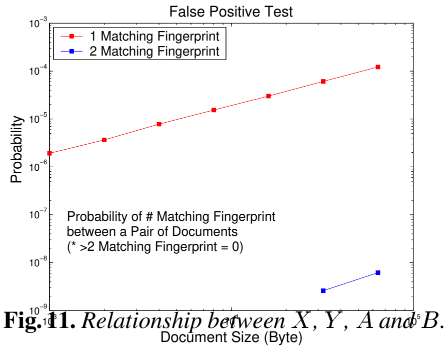

Fig. 11. *Relationship between X, Y, A and B.*

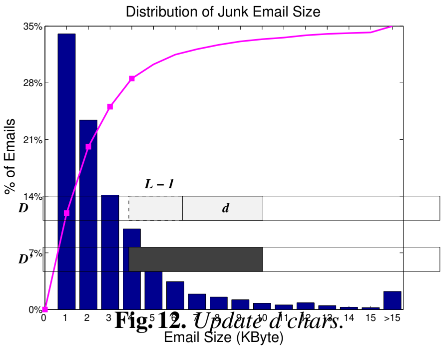

Fig. 12. *Update d chars.*

Fig. 8. *False Positive* probability of two random text files matching i ($i = 1, 2$) out of 10 fingerprint vectors, as a function of file size.

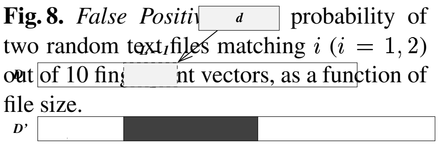

Fig. 13. *Insert d chars.*

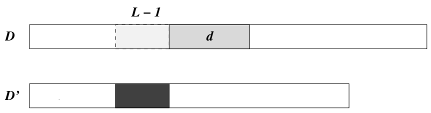

Fig. 14. *Delete d chars.*

A : the set of checksums calculated from all substrings in D
B : the set of checksums calculated from all substrings in D'
X : $A - B$, checksums from D which are not present in checksums of D'
Y : $B - A$, checksums from D' not present in original checksums of D
$FP(A)$: the fingerprint vector generated from checksums of D, such that $FP(A) \subseteq A$, $|FP(A)| = N$
$FP(B)$: the fingerprint vector generated from checksums of D', such that $FP(B) \subseteq B$, $|FP(B)| = N$
$|S|$: if S is a set or vector, $|S|$ represents the size of S
z : $|FP(B) - FP(A)|$, number of checksums in new fingerprint vector which are not in the old fingerprint vector

Refer to Figure 11 for an illustration of X, Y, A and B.

Let's define $P_r(x)$ as the probability that x out of N checksums in $FP(A)$ are obsolete, that is, not in B; define $P_r(y)$ as the probability that y out of N checksums in $FP(B)$ are newly generated, that is, not in A. We have:

$$P_r(x) = P_r(|FP(A) \cap X| = x) = \frac{\binom{|X|}{x} \times \binom{|A| - |X|}{N - x}}{\binom{|A|}{N}} \tag{1}$$

$$P_r(y) = P_r(|FP(B) \cap Y| = y) = \frac{\binom{|Y|}{y} \times \binom{|B| - |Y|}{N - y}}{\binom{|B|}{N}} \tag{2}$$

If:

1. $(N-x)+y<N$ (that is, $x>y$): $FP(B)$ is composed of $(N-x)$ checksums from $FP(A)$, y checksums from newly generated set Y, and others from $A\cap B$. That is, the y checksums from Y and others from $A\cap B$ are the new checksums in $FP(B)$ since $FP(A)$. Then, $z=N-(N-x)=x$.
2. $(N-x)+y\geq N$ (that is, $y\geq x$): $FP(B)$ is composed of y checksums from Y, other checksums from $FP(A)-X$. That is, the y checksums from Y are new checksums in $FP(B)$ since $FP(A)$. Then, $z=y$.

So, when $x>y$, $z=x$; when $y\geq x$, $z=y$. That is, $z=max(x,y)$. Then,

$$P_r(z)=P_r(y=z)\sum_{i=0}^{z}P_r(x=i)+P_r(x=z)\sum_{i=0}^{z}P_r(y=i)-P_r(x=z)P_r(y=z) \quad (3)$$

Let's define $P(|FP(A)\cap FP(B)|\geq k)$ to be the probability that at least k checksums are in common between fingerprint vector of new document and of old document. We have:

$$P_r(|FP(A)\cap FP(B)|\geq k)=P_r(|FP(B)-FP(A)|\leq N-k)=\sum_{i=0}^{N-k}P_r(z=i) \quad (4)$$

Knowing of $|X|$ and $|Y|$, we can apply results in equation (1)-(3) to equation (4), and then get the probability of the number of unchanged fingerprints after modification of the document.

While $|X|$ and $|Y|$ are related to modification pattern, we can further consider how to get $|X|$ and $|Y|$. $X=\bigcup X_i$ and $Y=\bigcup Y_i$, where X_i and Y_i are changes made to checksums because of one modification operation i.

We have three types of operations:

Update d characters : $|X_i|=L-1+d$, $|Y_i|=L-1+d$. This is illustrated in Figure 12.
Insert d characters : $|X_i|=L-1$, $|Y_i|=L-1+d$. This is illustrated in Figure 13.
Delete d characters : $|X_i|=L-1+d$, $|Y_i|=L-1$. This is illustrated in Figure 14.

X equals the union of each X_i and Y equals the union of each Y_i. So, if there is only one modification, we can exactly compute $|X|$ and $|Y|$. If there are more than one modification, $|X|$ ranges from $max_i|X_i|$ to $\sum_i|X_i|$, $|Y|$ ranges from $max_i|Y_i|$ to $\sum_i|Y_i|$. We can compute approximate average $|X|$ and $|Y|$ for a specific pattern of modification operations according to equations above.

Thus, we can use equation (4) to compute the probability distribution of number of unchanged fingerprints in fingerprint vector.