Eleventh IEEE Symposium on High Performance Distibuted Computing, Aug 2002, Edinburgh, Scotland, pp. 321–330

A Secure Distributed Search System

Yinglian Xie[†] David O'Hallaron[∗] Michael K. Reiter[∗]
[†]Department of Computer Science
[∗]Department of Computer Science and Department of Electrical and Computer Engineering
Carnegie Mellon University
Email:{ylxie, droh, reiter}@cs.cmu.edu

Abstract

This paper presents the design, implementation and evaluation of Mingle, *a secure distributed search system. Each participating host runs a Mingle server, which maintains an inverted index of the local file system. Users initiate peer-to-peer keyword searches by typing keywords to lightweight Mingle clients. Central to Mingle are its access control mechanisms and its insistence on user convenience. For access control, we introduce the idea of* access-right mapping, *which provides a convenient way for file owners to specify access permissions. Access control is supported through a single sign-on mechanism that allows users to conveniently establish their identity to Mingle servers, such that subsequent authentication occurs automatically, with minimal manual involvement. Preliminary performance evaluation suggests that Mingle is both feasible and scalable.*

1 Introduction

During the normal course of our work, we have managed to acquire accounts on dozens of different laptop, desktop, and remote hosts. Thousands of files are stored on the local disks of these hosts. With so many files on so many computers, it is becoming increasingly difficult and time consuming for us to find our own data and let other people find and share our data.

Although there are a number of useful tools for locating data on a single machine, locating data in a distributed environment is still troublesome. Tools like *grep* and *find* are good for searching small directory hierarchies, but are inappropriate for searching entire disks. The GNU *locate* command provides fast keyword search of file names, but not the contents of those files. Tools such as Glimpse [16] and Windows Indexing Service precompute inverted index tables of local files. Each entry in the index table stores a word and its occurrences in the files, which enables fast keyword querying. However, these tools do not support querying across different computers. Peer-to-peer applications such as Napster [18], Gnutella [8] and Freenet [3] have been used for large-scale locating and sharing of MP3 files. A serious drawback of such systems is that there is no security mechanism to protect data from access by unauthorized users. In addition, no indexing is provided to quickly locate information.

Thus, it would appear that we need new systems that help people find the data in their personal distributed computing environments. These systems should be both efficient, in the sense that our searches complete quickly, and secure, in the sense that unauthorized users are not allowed to locate our data. So how might we build such systems?

A straightforward solution is to build a global indexing service, where dedicated servers crawl files from every computer on the Internet and then compute a centralized index table. Search engines like Google [9] have used this kind of scheme very effectively for the Web. However, the centralized model is inappropriate for searching personal computing systems for a number of reasons. First, indexing is an expensive operation requiring large amounts of memory and disk space. Even massive search engines like Google can index only limited number of Web pages. Therefore, centralized indexing servers can not scale with the increasing number of computers and the exploding capacities of modern disks. Second, many personal files are private in nature. Users lose control of their files once they are indexed by the server. Even with complicated security and access control mechanisms, they may be unwilling to release their files to the dedicated servers.

Another approach is to have one or more dedicated indexing servers for a cluster of computers. For example, distributed search engines such as Harvest [1] and Oasis [19] set up one or more index servers to search within an intranet. With this scheme, a significant amount of network traffic is needed to fetch distributed files to the servers for index computing. More important, this scheme requires large, expensive, dedicated indexing servers, which are not feasible in a personal computing environment.

In this paper, we present Mingle, a secure distributed search system. Mingle is designed to meet the following requirements, which we consider fundamental to a distributed search system for personal computing:

- *Searches should be fast.* For fast search, Mingle precomputes an inverted index of local files on each participating host. A query can be processed by the local host, or routed through the participating hosts using peer-to-peer communication to locate all of the desired data.

- *The system should scale.* Since each participating host devotes computing resources for indexing, the Mingle scheme should scale well with the number of computers. Participating hosts communicate with each other only when there is a search request, greatly reducing network traffic.

- *The system should be secure:* The Mingle security architecture focuses on preventing unauthorized release of information while allowing files to be maximally shared. Since search is a frequent operation, we insist that the security mechanism be as convenient as possible for users. Access control policy is expressed using an *access-right mapping*, a novel mechanism that extends a local file system's access control primitives to Mingle users in a uniform and convenient way. This mechanism builds upon a single sign-on mechanism implemented in Mingle, which allows a user to perform authenticated search requests across many Mingle servers seamlessly.

The remainder of the paper is organized as follows. Section 2 summarizes related work. Section 3, 4, 5 are the core of the paper, describing the Mingle prototype, including the novel security architecture based on access-right mapping. Mingle is currently targeted for personal computing environments with tens of hosts. Preliminary performance evaluation of the Mingle prototype in Section 6 suggests that the system is feasible and scalable in such an environment.

2 Related Work

Distributed search has been studied in the area of information retrieval [2, 7, 12, 27], with emphasis on algorithms for server selection and result merging. Mingle is different from these works in that our focus is on the system architecture and the security mechanisms to prevent data from access by unauthorized users.

Distributed service and resource discovery [4, 10, 26] is a special type of distributed search, where queries consist of resource or service attributes instead of keywords. In such systems, search is performed at hierarchical directory servers in large scaled networks consisting of heterogeneous hosts. Compared with these systems, Mingle is currently targeted for a cluster of computers, and search is performed among friendly end hosts without centralized servers. Because of the architectural level differences, the security emphases are different. These systems assume malicious attacks and stress data privacy and integrity. Mingle focuses on access control and has a strong emphasis on user convenience while at the same time preserving file system access control semantics.

Peer-to-peer systems [21, 25, 24, 29] have been designed to locate objects in self-organizing overlay networks. Such systems use hash based distributed indexing schemes to locate objects. The location of each object is stored at one or more nodes selected by a distributed hash function. Although hash functions can deterministically locate object, they do not support keyword searching, a desirable operation to search information among personal data.

3 Overview of the Mingle System

Figure 1 shows the overall architecture of a Mingle cluster. On each host, there is a Mingle server running as a daemon. Communication among servers is peer-to-peer. A user may issue a request from any host to any of the servers by launching a lightweight client program, which simply sends the request to the local server and waits for replies. If only a local reply is required, then the local server handles the request and sends back the reply. Otherwise, the server forwards the request to remote servers for further processing.

Mingle clients issue separate requests for indexing and searching. Only the owners of files can issue requests to index those files. Any Mingle user can issue a search request to any Mingle server, but they only receive information about files that they are authorized to see, as described in the next section.

When a Mingle server daemon is started on a host for the first time, none of the files on that host are indexed. Users must make explicit requests to the Mingle server to index directory trees that they own. Thus Mingle is "opt-in", in the sense that users on Mingle hosts must issue explicit requests before their data is indexed and made available to Mingle clients.

Each Mingle server computes an inverted index of local files that have been indexed. The inverted index consists of: (1) a lexicon containing all of the words that appear in the files; and (2) an inverted file entry for each word, which stores a list of pointers to the occurrences of that word in the files. To locate a given word, only its inverted file entry needs to be traversed, allowing fast queries. The detailed design and data structures of the inverted index in Mingle are discussed in Section 5.1

In many cases, a user may wish to search all hosts in a Mingle cluster without specifying host identities. To enable this, we establish a *master server*, which is a normal Mingle server that maintains the list of host names inside the cluster. As shown in Figure 1, upon reception of a user request

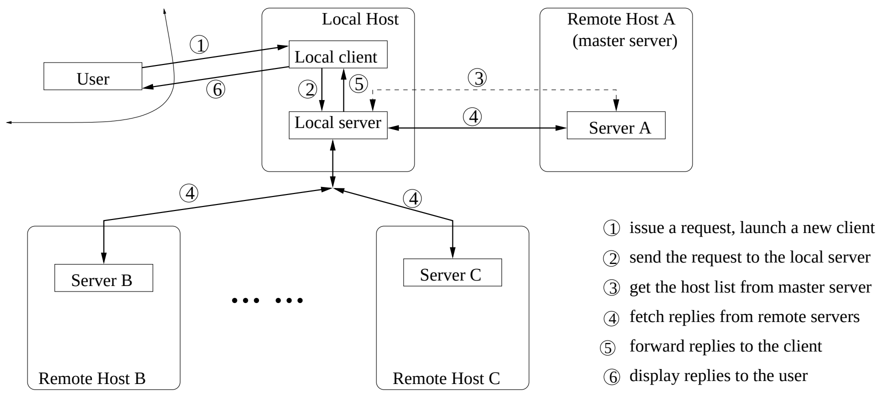

Local Host
Remote Host A (master server)
User
Local client
Local server
Server A
Server B
Server C
Remote Host B
Remote Host C
① issue a request, launch a new client
② send the request to the local server
③ get the host list from master server
④ fetch replies from remote servers
⑤ forward replies to the client
⑥ display replies to the user

Figure 1. Architecture of a Mingle cluster.

that needs to be routed through the cluster, the local server first fetches the host list from the master server, and then forwards the request to each remote host in the list individually.

While the server is implemented as a daemon, the client program is a lightweight program that is launched only when needed. An alternative design is for each user to run their own stand-alone server that they interact with via the command line. This approach would lead to multiple server processes running simultaneously on the same computer, leading to a large overhead. By implementing a separate lightweight client program, we build one index table and enable multiple users on the same host to share a single Mingle server.

4 Mingle Security Architecture

Since Mingle is used in personal computing environments where servers are assumed to trust each other, the Mingle security architecture focuses on preventing unauthorized release of information while allowing files to be shared among different types of users. Both the *local users* who have accounts on the Mingle host and *remote users* who do not have accounts should be able to participate in Mingle. Sensitive files can be accessed only by authorized users, while public files can be searched by even anonymous users.

Existing mechanisms are deficient in terms of both flexibility and user convenience. Mingle users might belong to different organizations and not have accounts on every machine. In this case, file system access controls on the Mingle hosts are not flexible enough to separate remote visitors into different classes of trustworthiness. Further, search is a stateless request involving one simple command. Supplying passwords with each request is not acceptable.

The design of the Mingle security architecture is guided by the following three principles:

- *File owners decide whom to trust.* We cannot expect every Mingle user to trust the same set of people. We must let file owners decide who is allowed to access their files.
- *Authorization is flexible and convenient.* Because some files are more sensitive than others and some people are more trustworthy than others, file owners must be able to specify access rights for different users on each single file conveniently.
- *Authentication has small overhead.* Since "search" is a stateless operation for everyday usage, we require the user authentication mechanism to be as lightweight as possible while providing reasonable level of security.

In the following, we present the details of the Mingle security architecture. We begin by describing the authorization mechanism, which addresses the first and second principles. We then discuss the user authentication mechanism, which addresses the third design principle. We close this section with a discussion of possible malicious attacks against Mingle.

4.1 Authorization

A straightforward way to handle access control is to maintain an access control list (ACL) for each file. Each item in the ACL specifies the permitted operations for each user. Although ACLs are flexible, they can be costly and prone to error since file owners must manually specify an ACL for each file.

We propose a new, more convenient approach that arises from the observation that the underlying file system in the Mingle host already enables access control on each file. By granting a user (or group) "read" permission to a file, the

```
// Return whether Mingle user U is al-
lowed to search file F
bool is_search_permitted(filename F, mingle_user U) {

    // Get the file owner of F
    O = get_file_owner(F);

    //Get the SPD of U with respect to the file owner O
    SPD = O.get_SPD(U);

    // Check if any member in SPD is allowed to read F
    foreach id in SPD {
        if F is readable by id {
            return true;
        }
    }

    return false;
}
```

Figure 2. The Mingle algorithm for access permission checking

file owner implicitly allows that user (or group) to search the file as well. However, the access control schemes in the file system are only applicable to local users who have accounts on the same computer.

To extend the file system access control to a remote user, we introduce the idea of *access-right mapping*. For a file owner with local account name *A*, a Mingle user with Mingle ID *U* (see Section 4.2) can be mapped to a *search protection domain (SPD)* that consists of one or more local users or user groups:

$$SPD_A(U) = \{userID1, userID2, \ldots, groupID1, groupID2, \ldots\},$$

where *userIDi* is some local user account name, and *groupIDi* is some local group name.

The meaning of the mapping is that Mingle user *U* has permission to search any local file owned by *A* and readable by one or more members of $SPD_A(U)$.

The process of access-right mapping is performed by each file owner independently. Thus, a Mingle user can be mapped to different SPDs by different file owners on the same host. Given the access-right mapping, the algorithm for access permission checking is simple, as shown in Figure 2.

The access-right mapping preserves the file system access control semantics. It greatly simplifies the access control specification, while giving file owners full control over their data. For example, each Unix file has 9 *mode* bits associated with it. These mode bits specify whether the file owner, specific group of users, and everyone else can read, write or execute the file. In many cases, a file owner can map a friendly remote Mingle user to an SPD that consists of only the owner account or a “guest” account, allowing file owners to specify access permissions to most of their files conveniently. For the small number of files that need fine-grained access control, file owners can define a user group for each file and map Mingle users to the corresponding user groups. In particular, in order to allow anonymous Mingle users to search shared public files, a file owner can define a special user group for public files and map any anonymous user to that group.

4.2 User Authentication

In Mingle, each request to index or search files on a host must be authenticated by that host. Since a Mingle user might not have accounts on every host, or might have different account names on different hosts, each Mingle user is assigned a unique global Mingle ID (a text string) that identifies the user to Mingle servers in a uniform way. A Mingle user without a Mingle ID is regarded as an anonymous Mingle user and can search only public files.

A Mingle ID is assigned to a user via a registration process that she executes once. In this registration process, the user selects and inputs a Mingle ID and password to her lightweight client, which conveys these inputs to the local Mingle server. The local Mingle server sends this pair to the master server for this Mingle cluster, using an encrypted channel (e.g., encrypted under the public key of the master server). The master server confirms that this Mingle ID has not previously been registered. If so, it generates a public signing key pair (e.g., [23]) for this Mingle ID, and saves the Mingle ID and associated password and key pair. Upon successful return, the user can convey her Mingle ID to other users in whatever way she wishes, so that these users can create access-right mappings (see Section 4.1) for this Mingle ID on other machines, as they choose.

This user can then execute distributed searches using Mingle from any computer running a Mingle server as follows. The user enters her Mingle ID, password, and search keyword into the lightweight Mingle client, which conveys these to the local Mingle server. The local Mingle server executes a protocol with the master server to retrieve the private key corresponding to this Mingle ID (using the password to authenticate to the master server). Once the private key is obtained, the local Mingle server can issue the query, containing the user’s Mingle ID and signed using the retrieved private key, to the relevant remote Mingle servers. Each remote Mingle server that receives this query can use the contained Mingle ID to retrieve the corresponding public key from the master server, and can then verify the digital signature using it.

There are numerous opportunities to use caching to eliminate steps in the above description and thereby improve the user experience. Specifically, the user’s local Mingle server can temporarily cache the user’s private key for use in subsequent searches, which eliminates the need for the user to re-enter her Mingle ID or password. Moreover, a remote Mingle server can temporarily cache the public key of this Mingle ID, so that it need not contact the master server again upon receiving another search query bearing this Mingle ID. Of course, this caching also introduces windows of vulnerability: e.g., if the user's public key is revoked due to the compromise of the corresponding private key, this may go unnoticed by a remote Mingle server that is caching the public key. It is therefore necessary to tune this caching to best balance performance, user experience, and security. Such tradeoffs are common in public key infrastructures (e.g., [14]), and we will not discuss them further in the present paper.

A benefit of this architecture is the fact that the user's password and private key are exposed only on machines where the user enters her password (and on the master server). Moreover, the protocol by which the user's machine retrieves the user's private key can be constructed to achieve strong security properties (e.g., see [20]), notably that the protocol messages themselves do not leak information that would permit an eavesdropping adversary to conduct a "dictionary attack" against the user's password [17, 13]. As a result, dictionary attacks are limited to online guesses sent to the master server, which the server can detect and stop. The primary vulnerability of this approach is the master server itself: if penetrated, the master server will leak all user's private keys. This risk can be mitigated by distributing the master server in a way that requires multiple master servers to be compromised to disclose sensitive data (e.g., [6]), though we have not implemented this approach in the present system.

We view the above approach to user authentication and single sign-on in Mingle as an interim solution suitable for small-scale Mingle deployments in user populations lacking a unified authentication infrastructure. For user populations with an existing authentication and single sign-on solution, ideally Mingle would exploit that solution for its user authentication needs, rather than "reinventing the wheel."

4.3 Other Vulnerabilities

Since Mingle assumes a friendly personal computing environment where servers trust each other, it is subject to various malicious attacks. Although we do not explicitly address how to defend against these attacks in Mingle, many of them can be prevented or mitigated by standard techniques. We briefly outline the types of malicious attacks that Mingle is vulnerable to and discuss possible ways to cope with them. Completely addressing these attacks is beyond the scope of this paper.

Mingle query responses are sent from remote servers unencrypted, and thus Mingle is vulnerable to information release and modification attacks. Moreover, without strong authentication of servers, a malicious Mingle server can provide fraudulent information. If data privacy and integrity is a major concern, then further cryptographic protocols can be used to authenticate servers as well as clients, and to set up session keys for message encryption.

A potential vulnerability to timing attacks exists within Mingle, due to its precomputation of an inverted index to permit fast searching. Specifically, the processing time for a Mingle server to compute its response is a function of the number of files actually containing the search item, not only those to which the client has search access. As a result, a client that can accurately measure the duration required for a Mingle daemon to respond to its search request can learn some information about the number of files on that host that contain the search item, even if the client has search access to very few of them. Randomizing search latencies could mitigate this threat. In addition, a filter could be applied to check user permission before searching through the inverted index. We note, however, that this threat applies only to files that their owners have volunteered to be indexed by Mingle.

Finally, like most other distributed systems, Mingle is vulnerable to various forms of denial-of-service attacks.

5 Mingle Implementation

In this section, we discuss the implementation of the Mingle server and client. We first describe the design of the inverted index in Mingle. Then we present the Mingle server architecture and explain the interactions among various system components.

5.1 Inverted Index

Indexing is a mechanism for quickly locating a given word in a collection of files. There are three common data structures for file indexing: *inverted index*, *signature files* and *bitmaps* (see [28]). An inverted index is the most natural indexing method, with each entry consisting of a word and its occurrences in the files. A signature file is a probabilistic method for file indexing, where each file has a signature. Every indexed word in a file is used to generate several hash values. The bits of the signature corresponding to those hash values are set to one, indicating the occurrences of the word. A bitmap stores a bit vector for every word. Each bit in the bit vector corresponds to a file and is set to one if the word appears in that file. Compared with the inverted index, signature files can cause false matches, resulting in either longer search times or large signature files. Bitmaps have relatively short search times, but require extravagant storage space and the update is slow when files are updated frequently. In Mingle, we decided to choose the inverted index because of its relatively small cost of storage and low search latency.

However, a fine-grained inverted index is still space consuming. A fine-grained inverted index containing all occurrences of every word can consume 50% to 300% of the original text size, which is not acceptable in personal computing. Therefore, Mingle computes a coarse-grained inverted index. Each index entry for a word contains only the first occurrence of that word in every file. A hash table is used to quickly locate the index entry for a word.

Word ID	Word	(Document ID; First occurrence)
1	movie	(1;6), (4;228)
2	day	(2;8), (3;57), (4;200)
3	event	(1;37), (3;22)

Figure 3. An example of inverted index table in Mingle

Before a file is indexed, it is assigned a document ID. Then the file is scanned word by word to build the index table incrementally. All of the words are converted to lower case. User specified stop words (defined by a configure file) are removed to reduce index size. For each word in the index table, if it appears multiple times in the file, then only the position of the first occurrence of that word will be recorded in the corresponding index entry. Figure 3 shows an example of inverted index table in Mingle. Once the index table is built, it can be updated regularly to remove out of date entries.

A query can consist of one or more keywords. With a coarse-grained inverted index table, queries are resolved in two steps. First, the corresponding index entries of the queried keywords are searched to return a list of files that match the query. Then, each individual file in the list is scanned to return all the exact occurrences of the queried keywords. For example, Figure 4 illustrates the search process for a query "movie AND event" using the index table in Figure 3. There is a tradeoff between index granularity and search latency. Compared with the fine-grained inverted index, a coarse-grained index table requires longer search latency since the second step will be otherwise unnecessary. However, the extra latency is typically small, as is shown in Section 6.1.

5.2 Mingle Server Architecture

The Mingle server is implemented as a single process (Figure 5). The *file descriptor manager* uses the `select` function to multiplex concurrent requests. After a request has been received by the *receiver*, it is parsed by the *request manager*, which determines the request type and forwards

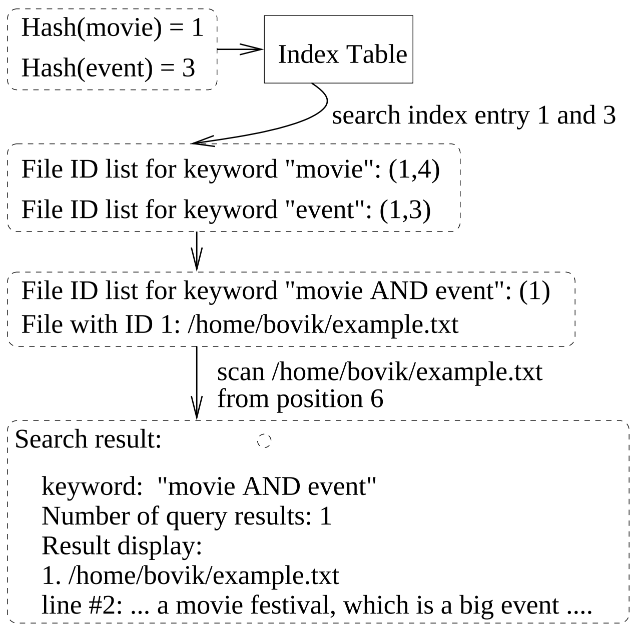

Hash(movie) = 1
Hash(event) = 3
Index Table
search index entry 1 and 3
File ID list for keyword "movie": (1,4)
File ID list for keyword "event": (1,3)
File ID list for keyword "movie AND event": (1)
File with ID 1: /home/bovik/example.txt
scan /home/bovik/example.txt
from position 6
Search result:
keyword: "movie AND event"
Number of query results: 1
Result display:
1. /home/bovik/example.txt
line #2: ... a movie festival, which is a big event ....

Figure 4. A query example in Mingle

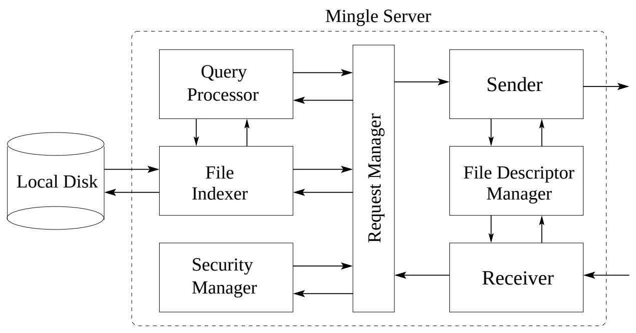


Figure 5. Mingle server architecture

the request to the appropriate components for further processing. The major components that process a user request are the *file indexer*, *query processor*, and *security manager*. The *file indexer* accesses files from the local disk and builds up an inverted index table in disk. For performance optimization, the *file indexer* maintains a cache in memory for frequently accessed terms and their indices. The *query processor* processes user queries, including advanced query options based on the index table built by the *file indexer*. The *security manager* performs security operations, including access control and user authentication. After the request processing is finished, the *sender* sends out the reply passed by the *request manager*.

Both the Mingle server and the client program are implemented in C++ in Linux. The request signing and signature verification use the RSA algorithm [23], which is implemented by the Crypto++ library (version 4.2) [5]. The communications among servers are via TCP connections, while the server and the client program communicate via Unix IPC.

6 Performance Evaluation

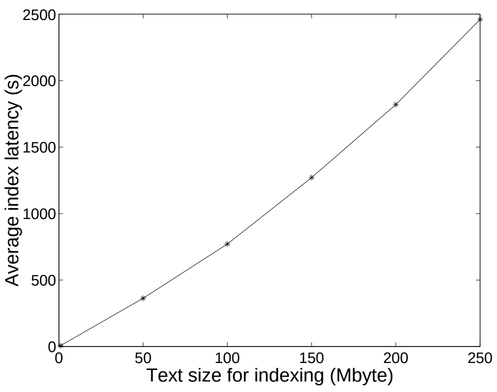
In this section, we present the performance evaluation of Mingle. We have conducted three sets of experiments to answer the following three questions: (1) What is the cost of index and search — the two major operations in Mingle? (2) What is the impact of our security mechanism on performance? (3) What is the scalability of Mingle? The first and the second sets of experiments are conducted on PIII 550MHz machines with 128 MB of RAM. The last set of experiments are run on the cluster of computers (PIII 550MHz) in a 10BaseT Ethernet LAN. Each data point in the figures is the average of ten runs.

6.1 What is the Cost of Index and Search?

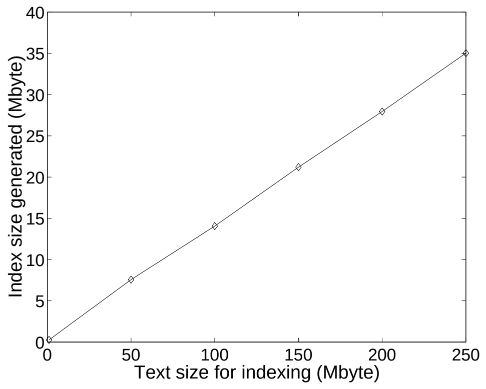
Since only text files will be indexed, we have downloaded the RFC [22] and the Internet Drafts [11] repositories to test the index performance. We vary the text size to be indexed. Figure 6 plots the index latency and the generated index table size. We observe that both costs increase linearly with the text size. It takes about 30 minutes to index 200 MB of text (about 9 seconds per 1 MB). Usually, only a portion of the data on a disk will be text. With the current index speed, we can index a local disk regularly at machine idle time. The generated index table size is about 15% of the original text size. Both costs are acceptable for personal computing.

With the pre-computed index table, we then examine the search latency on the local server without using our security mechanism. We vary the indexed text size and the number of keywords in a query. Figure 7(a) plots the query lookup latency in case of cache hits, when the required items in the index table are already in memory. Overall, the search latency is on the order of milliseconds and seconds, which is fast. For example, in 200 MB text, it takes about 250 ms to find answers to a two-keyword query, while it takes as long as 15 seconds to get the same results using "grep". If the keywords in a query do not exist in the indexed text, the search latency is less than 1ms regardless of the indexed text size.

We are also interested in the penalty of a cache miss, when the required items in the index table need to be fetched from the disk. Figure 7(b) shows the comparison of search latency in case of cache hit and cache miss. In both cases, the indexed text size is 100 MB. As indicated in the figure, the penalty of a cache miss is on the order of tens of milliseconds, which is relatively small.



(a) Index Latency



(b) Index Size

Figure 6. Index latency and disk size vs. text size indexed

6.2 What is the Impact of Security on Performance?

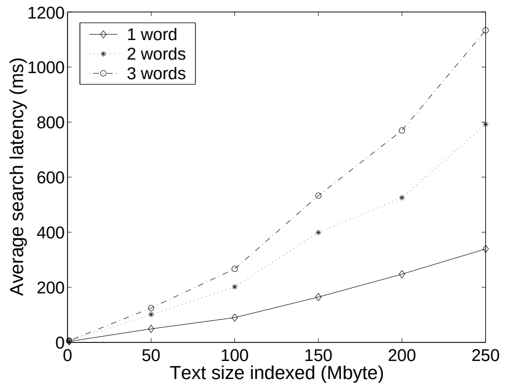
In this section, we measure the impact of the Mingle security mechanism on performance. Since cryptographic computation is often expensive, our main concern is the latency penalty of cryptographic operations for remote user authentication. We evaluate the cost of request signing and signature verification by measuring the time spent in each step of request processing.

We conducted our experiments on two machines serving as the local server and the remote server respectively in the same LAN. Since the security penalty does not depend on request type, we choose a 3-keyword search request as our example and fix the indexed text size to be 100 MB. We use 1024-bit and 512-bit [1] RSA keys, respectively. Figure 8 lists the processing steps we are interested in. The pro-

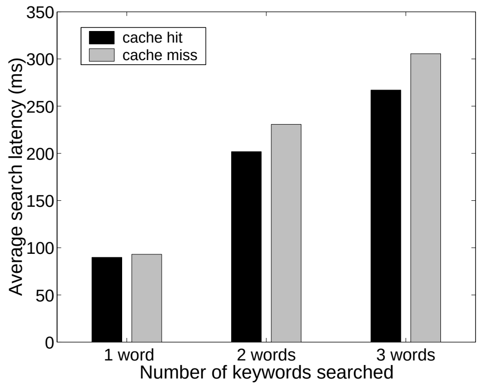
[1]Each user can select her own key length in Mingle. Though we report performance for 512-bit keys here, such keys provide insufficient security for commercial applications [15] and are discouraged for use in any commercial application of Mingle.

	Total	Parsing	Networking	Look up	Signing	Sig.Check
Mean (1024 bit)	313590	940	6010	279530	25710	1400
Std dev (1024 bit)	2615	15	643	2613	44	25
Percentage (1024 bit)	100.0%	0.3%	1.9%	89.1%	8.2%	0.5%
Mean (512 bit)	292440	950	5610	279650	5390	850
Std dev (512 bit)	1100	10	55	1037	18	10
Percentage (512 bit)	100.0%	0.3%	1.9%	95.6%	1.9%	0.3%

Figure 8. Time to process a search request using 1024-bit and 512-bit RSA keys (μs).



(a) Search latency vs. text size indexed



(b) Search latency with cache hit/cache miss

Figure 7. Search latency on a single server

cessing consists of two stages: First, the request is parsed and signed at the local server, and forwarded to the remote server. Second, the remote server verifies the signature and generates the reply by query lookup. The "Total" column corresponds to the time elapsed between the arrival of the request and sending the reply to the client program by the local server. The "Networking" column corresponds to the latency spent in forwarding the request and getting the reply from the remote server. For each step, we show the mean and the standard deviation of latency as well as the percentage of total latency.

We can see from Figure 8 that most of the processing latency is spent on query lookup. Although request signing is also expensive, it is not the performance bottleneck. Compared with signing, signature verification is fast. Note that the standard deviation is small for all steps except networking latency, which has a relatively larger variation due to the network instability. In summary, our security mechanism has little impact on overall search performance.

6.3 What is the Scalability of Mingle?

In this section, we examine whether Mingle is able to scale with an increasing number of hosts. We consider scenarios with and without our security mechanism. We run the Mingle server on every host in a cluster of up to 23 computers. Each server has a precomputed index table of 100 MB text.

Figure 9(a) plots the average search latency and the standard deviations without security checking by varying the number of participating hosts. We can see that the performance degradation is not constant with the increasing number of Mingle servers. The search latency increases most when the number of hosts in Mingle increases from one to three. The increased latency is due to network communication and remote processing, which do not happen in the single server case. When we further increase the number of the participating hosts, the performance degradation becomes smaller. The reason is that although the network communication time is increased, the remote processing can be done in parallel on different servers. We observe that the search latency has higher standard deviation with the increased number of servers. This is because the network latency variation increases with the number of hosts in the system. If security checking is enforced, the overall search latencies increase only slightly, as indicated by Figure 9(b). We note that when the number of hosts is greater than 21, the search latency with security checking is even lower than that without security checking. This is because the security overhead is small compared with the overall search latency. The lower search latency with security checking is due to the large variance of network latencies when there are more hosts in the cluster. Overall, our measurements suggest that

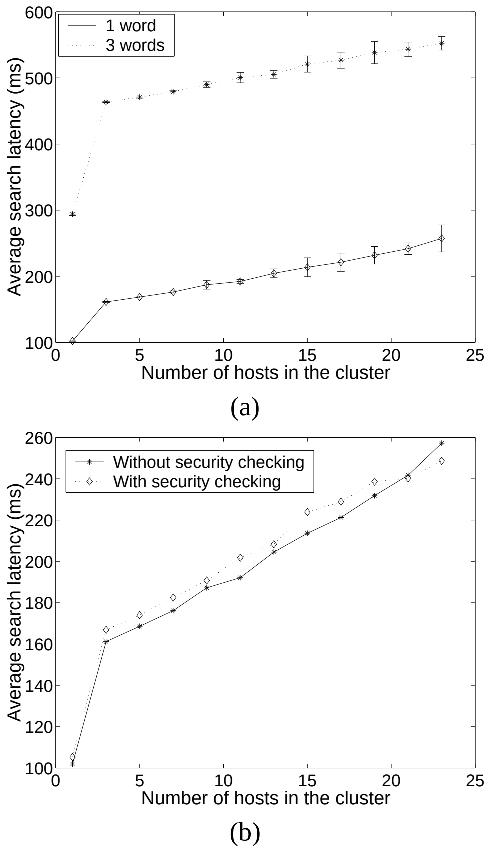


Figure 9. Search latency in a Mingle cluster. (a) Search latency without security checking (b) Search latency with and without security checking

Mingle is able to scale with increasing number of hosts.

7 Concluding Remarks

We have developed Mingle to help authorized users efficiently locate their personal data on distributed computers. Mingle hosts precompute inverted index of local files, searching among each other in a peer-to-peer way. The Mingle security architecture consists of authorization and authentication mechanisms. One of the major benefits of our security mechanism is user convenience. For authorization, we introduce an access-right mapping that allows data owners to conveniently specify access permissions. This is supported using a user authentication mechanism that permits a form of single sign-on.

Preliminary performance evaluation of Mingle suggests that: (1) Both the cost of index and search grow linearly with the indexed text size. (2) The Mingle security mechanism has little impact on search performance. (3) Mingle is able to scale with increasing number of hosts.

Future work includes expanding Mingle to larger networks, considering schemes for encrypting and replicating host indexes, and better understanding Mingle's vulnerability to attacks such as timing attacks.

8 Acknowledgements

We are grateful to Srinivasan Seshan, Yang-hua Chu, Julio Lopez, Hyang-Ah Kim and Tiankai Tu for their comments and suggestions. We also thank the anonymous reviewers for their comments. This work is sponsored in part by the National Science Foundation under Grant CMS-9980063, and in part by a grant from the Intel Corporation.

References

[1] C. M. Bowman, P. B. Danzig, D. R. Hardy, U. Manber, and M. F. Schwartz. Harvest: A Scalable, Customizable Discovery and Access System. Technical Report CU-CS-732-94, Department of Computer Science, University of Colorado, Boulder, 1994.

[2] J. P. Callan, Z. Lu, and W. B. Croft. Searching Distributed Collections with Inference Networks . In *Proceedings of the 18th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 21–28, Seattle, Washington, 1995. ACM Press.

[3] I. Clarke, O. Sandberg, B. Wiley, and T. W. Hong. Freenet: A Distributed Anonymous Information Storage and Retrieval System. In *Designing Privacy Enhancing Technologies:International Workshop on Design Issues in Anonymity and Unobservability, LNCS 2000*, December 2000.

[4] S. E. Czerwinski, B. Y. Zhao, T. D. Hodes, A. D. Joseph, and R. H. Katz. An architecture for a secure service discovery service. In *Mobile Computing and Networking*, pages 24–35, 1999.

[5] W. Dai. Crypto++. http://www.eskimo.com/˜weidai/cryptlib.html.

[6] W. Ford and B. Kaliski. Server-Assisted Generation of a Strong Secret from a Password. In *Proceedings of the IEEE 9th International Workshops on Enabling Technologies: Infrastructure for Collaborative Enterprises, NIST, Gaithersburg MD*, June 2000.

[7] J. French, A. Powell, J. Callan, C. Viles, T. Emmitt, K. Prey, and Y.Mou. Comparing the performance of database selection algorithms. In *Proceedings of the 22nd Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 238–245, 1999.

[8] Gnutella hosts. http://www.gnutellahosts.com.

[9] Google. http://www.google.com.

[10] E. Guttman, C. Perkins, J. Veizades, and M. Day. Service location protocol, version 2. *IETF. RFC 2165*, November 1998.

[11] Internet-Drafts. http://www.ietf.org/ID.html.

[12] S. Kirsch. Document retrieval over networks wherein ranking and relevance scores are computed at the client for multiple database documents. *U.S.Patent 5,659,732*, 1997.

[13] D. Klein. Foiling the cracker: A survey of, and improvements to, password security. In *2nd USENIX Security Workshop*, August 1990.

[14] B. Lampson, M. Abadi, M. Burrows, and E. Wobber. Authentication in distributed systems: Theory and practice. In *ACM Trans. Computer Systems 10, 4*, pages 265–310, November 1992.

[15] A. Lenstra and E. Verheul. Selecting cryptographic key sizes. In *Proceedings of the 2000 International Workshop on Practice and Theory in Public Key Cryptography (PKC)*, January 2000.

[16] U. Manber and S. Wu. GLIMPSE: A Tool to Search Through Entire File Systems. Technical Report 34, Department of Computer Science, The University of Arizona, 1993.

[17] R. Morris and K. Thompson. Password security: A case history. In *Communications of the ACM, 22(11)*, pages 594–597, November 1979.

[18] Napster. http://www.napster.com.

[19] Oasis. http://www.oasis-europe.org.

[20] R. Perlman and C. Kaufman. Secure Password-Based Protocol for Downloading a Private Key. In *Proceedings of the 1999 Network and Distributed System Security*, February 1999.

[21] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker. A Scalable Content-Addressable Network. In *Proceedings of ACM Sigcomm*, August 2001.

[22] RFC. http://www.rfc-editor.org.

[23] R. L. Rivest, A. Shamir, and L. M. Adleman. A method for obtaining digital signatures and public-key crypt osystems. *Communications of the ACM*, 27(2), February 1978.

[24] A. Rowstron and D. P. Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems. In *IFIP/ACM International Conference on Distributed Systems Platforms (Middleware), Heidelberg, Germany*, pages 329–350, November 2001.

[25] I. Stoica, R. Morris, D. Karger, M. Kaashoek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for Internet applications. In *Proceedings of ACM Sigcomm*, August 2001.

[26] M. van Steen, F. Hauck, P. Homburg, and A. Tanenbaum. Locating Objects in Wide Area Systems. In *IEEE Communications Magazine*, pages 104–109, 1998.

[27] C. L. Viles and J. C. French. Dissemination of collection wide information in a distributed information retrieval system. In *Proceedings of the 18th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 12–20, 1995.

[28] I. WITTEN, A. MOFFAT, and T. Bell. Managing gigabytes: Compressing and indexing documents and images. *Second ed. Morgan Kaufmann*, 1999.

[29] Y. Zhao, J. D. Kubiatowicz, and A. Joseph. Tapestry: An Infrastructure for Fault-tolerant Wide-area Location and Routing. Technical Report UCB//CSD-01-1141, U. C. Berkeley, April 2000.