# ConChord: Cooperative SDSI Certificate Storage and Name Resolution

Sameer Ajmani, Dwaine E. Clarke, Chuang-Hue Moh, and Steven Richman[⋆]

MIT Laboratory for Computer Science
200 Technology Square, Cambridge, MA 02139, USA
{ajmani,declarke,chmoh,richman}@lcs.mit.edu

**Abstract.** We present ConChord, a large-scale certificate distribution system built on a peer-to-peer distributed hash table. ConChord provides load-balanced storage while eliminating many of the administrative difficulties of traditional, hierarchical server architectures.
ConChord is specifically designed to support SDSI, a fully-decentralized public key infrastructure that allows principals to define local names and link their namespaces to delegate trust. We discuss the particular challenges ConChord must address to support SDSI efficiently, and we present novel algorithms and distributed data structures to address them. Experiments show that our techniques are effective and practical for large SDSI name hierarchies.

## 1 Introduction

SDSI (Simple Distributed Security Infrastructure) [19] is a proposed public key infrastructure that is more powerful and flexible than existing systems like DNS-EXT [7] and X.509 [17]. In SDSI, names are defined in local namespaces, and longer names can link multiple namespaces to delegate trust. This design obviates central certification authorities, allowing principals to declare and modify complex trust relationships.

For example, suppose Acme wants to allow access to their web site only to their partner companies' employees. SDSI allows Acme to define the group "Acme's partners" and delegate trust to each partner to define their own group of "employees." Acme's web server can enforce the access control policy by requiring that each HTTP client prove membership in the group "Acme's partners' employees." A client satisfies this requirement by presenting two certificates: one that shows that she is the "employee" of a company, and another that shows that her company is a "partner" of Acme.

Locating the certificates that a client needs is simple when certificates are stored at a central server, but this defeats the purpose of SDSI's decentralized design and scales poorly. We could distribute certificate storage using a server hierarchy, like DNS. However, unlike DNS, SDSI has no single root, and so requires some non-hierarchical way to locate the server that stores a certificate.

[⋆] Authors in alphabetical order.

The SPKI/SDSI IETF working group suggests embedding URIs in public keys for this purpose [9], but this seems undesirable, as changes to a key's URI invalidate certificates issued for that key. Also, since one SDSI name can be defined in terms of another, SDSI name resolution is fundamentally more complex than DNS name resolution. Certificates from many different organizations may be required to create a proof, and it is not always clear which organization should store a partial or completed proof.

Server hierarchies also suffer from administrative problems. A large fraction of DNS traffic is caused by "misconfiguration and faulty implementation of the name servers" [5]. Making such systems fault-tolerant requires even more expertise and resources.

To address these challenges, we present ConChord,[1] a distributed SDSI certificate directory built on a peer-to-peer storage system. Peer-to-peer systems [6, 8] configure themselves to provide immense storage capacity, high reliability, balanced load, and efficient lookups. ConChord uses the Chord [22] lookup system, with storage and caching techniques based on the Cooperative File System (CFS) [6]. ConChord locates certificates using relevant information (such as the name a certificate resolves), eliminating the need to embed URIs in public keys.

ConChord supports three operations: inserting a new certificate, resolving a name, and checking whether a name resolves to a specific key. ConChord's prototype implementation supports these operations, but does not yet support replication or recovery from network partitions. ConChord's current design does not handle server failures, restrict access to certificates, enforce storage quotas, or resist malicious attacks; these issues are left for future work.

The rest of this paper is organized as follows: Section 2 describes the capabilities and semantic richness of the SDSI naming system. Section 3 presents ConChord's data structures, algorithms, and storage design, and Section 4 presents a brief evaluation. Section 5 discusses related work, and Section 6 concludes.

## 2 SDSI Background

The main innovation of SDSI is the use of *local names*. Unlike DNS, in which names must be unique in a global namespace, a SDSI name has meaning relative to the principal defining that name. For instance, Professor X and Professor Y can each define the name "RAs" to refer to their respective research assistants. The two groups of RAs are referred to by the local names "$K_{ProfX}$ RAs" and "$K_{ProfY}$ RAs", where $K_P$ is principal P's public key. In a system that uses SDSI for authorization, Professor X might add "$K_{ProfX}$ RAs" to the access control list for a file, effectively stating that her RAs are the only principals allowed to access that file.

Principals define local names with two kinds of cryptographically-signed certificates: *reducing* and *non-reducing* [4]. A *reducing* certificate binds a local name to a principal. So, if Professor X wants to add Bob and Carol to her group of

[1] Certificates on Chord

RAs, she can issue two reducing certificates:

$$\begin{aligned} K_{\mathit{ProfX}} \text{ RAs} &\longrightarrow K_{\mathit{Bob}} \\ K_{\mathit{ProfX}} \text{ RAs} &\longrightarrow K_{\mathit{Carol}} \end{aligned} \tag{1}$$

The *value* of a SDSI name is the union of all keys that are bound to it, so here the value of the name "$K_{\mathit{ProfX}}$ RAs" is the set $\{K_{\mathit{Bob}}, K_{\mathit{Carol}}\}$.

We call the operation that returns the value of a name *name resolution*, or simply resolution. We call the operation that verifies that a specified principal is in the set of keys bound to a name *membership checking*. Finally, we call the issuance of a new SDSI certificate *insertion*.

Although certificates can only be issued for local names (which have exactly one string component), resolutions and membership checks can be carried out for longer *extended names*. For instance, if MIT issues the certificate

$$K_{\mathit{MIT}} \text{ faculty} \longrightarrow K_{\mathit{ProfX}} \tag{2}$$

then we can resolve the extended name "$K_{\mathit{MIT}}$ faculty RAs". Semantically, this name denotes all principals that have been designated as RAs by all principals designated as MIT faculty. Given the above certificates, this name resolves to the set $\{K_{\mathit{Bob}}, K_{\mathit{Carol}}\}$. If MIT also issued the certificate "$K_{\mathit{MIT}}$ faculty" $\longrightarrow$ $K_{\mathit{ProfY}}$, then "$K_{\mathit{MIT}}$ faculty RAs" would also include Professor Y's RAs. Bob can prove that he is a member of "$K_{\mathit{MIT}}$ faculty RAs" by presenting the *sequence* of certificates (2)(1); anyone can verify this proof by checking the signatures on the two certificates.

The second type of certificate is the *non-reducing* certificate, which binds a local name to another (local or extended) name:

$$\begin{aligned} K_{\mathit{MIT}} \text{ staff} &\longrightarrow K_{\mathit{MIT}} \text{ faculty} \\ K_{\mathit{MIT}} \text{ staff} &\longrightarrow K_{\mathit{MIT}} \text{ faculty RAs} \\ K_{\mathit{MIT}} \text{ staff} &\longrightarrow K_{\mathit{HR}} \text{ visiting} \end{aligned} \tag{3}$$

Notice that the right-hand sides (called *subjects*) of the above non-reducing certificates are names, whereas the subjects of reducing certificates are principals' keys. A non-reducing certificate states that the value of a local name includes the value of the subject. So, given these certificates, we can resolve "$K_{\mathit{MIT}}$ staff" as the union of the values of "$K_{\mathit{MIT}}$ faculty", "$K_{\mathit{MIT}}$ faculty RAs", and "$K_{\mathit{HR}}$ visiting".

Since the name bound by reducing certificate (2), "$K_{\mathit{MIT}}$ faculty", is a prefix of the subject of non-reducing certificate (3), these certificates are called *compatible*, and we can *compose* (3) with (2) to yield a new, derived certificate:

$$K_{\mathit{MIT}} \text{ staff} \longrightarrow K_{\mathit{ProfX}} \text{ RAs} \tag{4}$$

This new certificate doesn't introduce any new trust relationships. Rather, it represents a trust relationship that already exists (we can use the original, signed certificates to prove this fact).

If we repeatedly perform all possible compositions over a certificate set until no more compositions are possible, we eventually have a set of reducing certificates that directly bind each local name to each key in that name's value. We call such a set *closed* under name-reduction. This closure is important for supporting efficient name resolutions and membership checks.

## 3 Design

ConChord's key design assumption is that membership checking is by far the most common operation on SDSI names, followed by name resolution. Insertion is comparatively rare. Accordingly, ConChord maintains closure over its certificates on each insertion, thereby reducing the amount of work required for name resolution and membership checking. Users can thus accelerate resolutions and checks for extended names by inserting non-reducing certificates.

ConChord's algorithms use three hash tables: check, value, and compatible (proposed in [10]). These tables are summarized in Table 1.

**Membership Checking** Every certificate inserted into ConChord is stored in the check table, where the hash key for each certificate $c$ is a function applied to the tuple $\langle$*c's name, c's subject*$\rangle$. If multiple certificates that bind the same name to the same subject are inserted into the check table, then the certificate with the latest expiration time overwrites the others.

To check whether a key $K$ is bound to name $n$, we can resolve $n$ and check whether $K$ is in the resulting set. If $n$ is a local name (like "$K_{MIT}$ staff"), then the closure property guarantees that the binding from $n$ to $K$ (if one exists) is already in the check table, so we can instead fetch $\langle n, K\rangle$ directly from check.

**Name Resolution** For each local name bound by a certificate, value stores the set of keys bound to that name. The hash key for value is a function of the name.

To resolve a local name, we just look it up in value. To resolve an extended name, we look up the value of the name's *prefix* (the prefix of an extended name "$K\ n_1 \ldots n_m$" is the local name "$K\ n_1$"), then we recursively resolve the rest of the name. For instance, to resolve "$K_{MIT}$ staff spouse", we first fetch the

**Table 1.** ConChord Hash Tables

| Table | Index | Value |
|---|---|---|
| check | name, subject | an entry whose *name* is name and whose *subject* is subject |
| value | name | a set of entries whose *name* is name and whose *subject* is a public key |
| compatible | name | a set of entries whose *subject* is a name that starts with name |

value for "$K_{MIT}$ staff". Then, for each staff member $K_S$, we fetch the value for "$K_S$ spouse" and take the union of those values to compute the result.

**Insertion** The above algorithms rely on two invariants. First, check and value are both up to date with respect to each other (if a name binding is in value, then the corresponding certificate is in check, and vice versa). We maintain this invariant by updating both tables when new certificates are inserted.

Second, closure is always maintained over the certificates. To maintain this invariant, we compose each new certificate with each other compatible certificate in the system. We then recursively insert the resulting derived certificates, since they may trigger further compositions.

When a new non-reducing certificate is inserted, we locate all compatible reducing certificates by looking up the prefix of the new certificate's subject in the value table. When a new reducing certificate is inserted, we must locate all compatible non-reducing certificates. To make this fast, ConChord maintains a third table, compatible, that stores non-reducing certificates, where the hash key of a certificate is a function of the prefix of its subject.

**Maintaining Proofs** We have said that check stores certificates, value stores keys, and compatible stores non-reducing certificates. In reality, these tables store *entries*, which are proofs of name bindings, and a single proof might consist of a sequence of certificates (if the binding was derived from a composition).

An entry consists of a *name*, a *subject*, and a certificate *sequence* that proves that the *name* is bound to the *subject*. For example, the entry for the derived certificate (4) would be:

$$\begin{aligned} \textit{name} &= K_{MIT} \text{ staff} \\ \textit{subject} &= K_{ProfX} \text{ RAs} \\ \textit{sequence} &= (3), (3)_{K^{-1}_{MIT}}, (2), (2)_{K^{-1}_{MIT}} \end{aligned}$$

where $X_{K^{-1}}$ represents the digital signature of $X$ using $K^{-1}$, $K$'s private key.

Like certificates, entries can be composed, in which case their sequences are concatenated. The expiration time of an entry is the earliest expiration of any certificate in its sequence.

Figure 1 presents the complete insertion algorithm using entries.

## 3.1 Peer-to-Peer Architecture

ConChord locates entries on servers using the Chord [22] distributed lookup system.[2] ConChord distributes its hash tables by mapping each hash key to a *Chord ID*. Clients access the hash tables by calculating the Chord ID for each hash key and contacting the appropriate server.

[2] ConChord could also use CAN [18], Pastry[20], or Tapestry[23].

```
insert(certificate c)
    entry e
    e.name ← K n  (the name bound by c)
    e.subject ← c's subject
    e.sequence ← $c, c_{K^{-1}}$
    insert(e)

insert(entry e)
    if (check[e.name, e.subject] is empty)
        check[e.name, e.subject] ← e
        if (e.subject is a public key)
            value[e.name] ← value[e.name] $\cup$ {e}
            set ← compatible[e.name]
            for each $e' \in$ set
                insert(compose(e', e))
        else
            compatible[prefix(e.subject)]
              ← compatible[prefix(e.subject)] $\cup$ {e}
            set ← value[prefix(e.subject)]
            for each $e' \in$ set
                insert(compose(e,e'))
    else if (check[e.name, e.subject] expires before e)
        check[e.name, e.subject] ← e

// requires $e_1$.subject = $e_2$.name $\cdot$ X
// for some (possibly empty) sequence of strings X
// returns the composed entry e
compose(entry $e_1$, entry $e_2$)
    e.name ← $e_1$.name
    e.subject ← $e_2$.subject $\cdot$ X
    e.sequence ← $e_1$.sequence $\cdot$ $e_2$.sequence
    return e
```

**Fig. 1.** Insertion with closure

A problem, however, arises with maintaining our invariants on each insertion. The first invariant (if an entry is in check, it is also in value or compatible, and vice versa) might be violated if a client crashes during an insertion. The second invariant (closure is maintained over the certificate set) might be violated if two compatible certificates are inserted concurrently or if a client crashes before inserting all derived certificates.

We could solve these problems using synchronization to provide transactional consistency for insertions; however, this is slow in the wide area. Instead, we allow the system to temporarily violate our invariants in the rare case that a problem occurs. To restore consistency, each server periodically reinserts the check entries it stores, so all compositions eventually happen. This is an efficient solution because the work of reinsertion is spread among the servers, and reinsertions can be infrequent.

Allowing such temporary inconsistencies safe with respect to security; they can only make some entries temporarily unavailable. Since SPKI/SDSI semantics are monotonic, the inability to locate some certificates cannot grant undeserved authority [12].

**Storage Details** The value and compatible tables store sets of entries, rather than single entries. A very large set (such as the value of "$K_{USA}$ citizens") might cause load imbalance or even exceed the capacity of a single server. Therefore, ConChord distributes entries in a set among several servers.

We might consider using CFS-style Merkle trees to distribute large data sets [6], but such data structures do not support concurrent modification by multiple clients. Because ConChord periodically reinserts entries and garbage-collects expired entries, sets must support concurrent modification. To do so, ConChord distributes the elements of a set over many servers, but serializes set modifications through a single server.

The members of the set whose Chord ID is $s$ are stored at Chord IDs $hash(s,\ 1) \ldots hash(s,\ T)$, where $T$ is the size of the set. The value of $T$ is stored as a *set size record* at ID $s$. Servers support two atomic operations on set size records: *get-size* and *increment-and-get*.

To fetch the members of a set, a client calls *get-size*, calculates the Chord IDs for all of the set's members, and retrieves them in parallel. To optimize for singleton sets, the client fetches the first entry of a set in parallel with the size.

To add a new entry to set $s$, a client first calls *increment-and-get*. This increments $T$ and returns the updated size, $T'$. The client then stores the new entry at ID $hash(s,\ T')$.

When an entry $e$ in a set expires, the server storing $e$ first looks for updated versions of the expired certificates in the check table. If no new certificates are found, the server storing $e$ tells the set size server that $e$ is no longer valid. The set size server compacts the set by fetching the last element in the set ($e'$), overwriting $e$ with $e'$, and decrementing the set size. As an optimization, the set size server can instead direct the next set insertion to overwrite $e$.

Recall that servers periodically reinsert entries; this involves (1) checking that each entry appears in the appropriate value or compatible set, and (2) checking that each entry is composed with all other compatible entries. The first check involves scanning the appropriate set; once done, this check need not be repeated. However, the second check needs to be repeated indefinitely in case new compatible entries are added. In the common case, the set of compatible entries will be unchanged from the previous reinsertion. To make checking for changes fast, we store a version number alongside each set size record. The version number is incremented each time an element is added to the set. Thus, a reinsertion usually only needs to check that the version number is unchanged, which is a single Chord lookup.

**Network Partitions** If a network partition splits the set of ConChord servers, servers in different partitions may store different values for the same Chord ID. When the partition heals, ConChord automatically resolves such inconsistencies. The server responsible for storing a set entry (in the healed partition) temporarily stores all entries accepted for that ID and lazily diverts all but one entry to the end of the set. Similarly, the server responsible for a set size record temporarily accepts the maximum size value proposed by any server, and lazily corrects the size (if necessary).

**Load Imbalance** To balance request load for popular entries, ConChord caches entries along lookup paths, as in CFS. Cached entries are expunged from a full cache in LRU order. Cached copies may become out-of-date, so servers assign them time-to-live values. Cached set size records have small TTLs, since insertions and compactions change the actual size values. Cached set entries have relatively larger TTLs, since they only become invalid when a server temporarily stores multiple entries (after a partition heals) or when a set is compacted after an expiration.

Clients and servers can detect and recover from out-of-date set size records by fetching past the expected end of the set until no more entries are found (locations *T+1*, *T+2*, etc.[3]). Set modifications cannot use cached set size records; they must update the original record.

## 3.2 Accelerating the Operations

Resolution of an extended name requires a value lookup for each part of the name, so resolution latency scales with name length. To reduce the number of lookups, we allow clients to *share resolutions* by caching extended name resolutions in the value and check tables. Then, clients can use cached prefixes when resolving a name. For example, if "$K_{MIT}$ faculty assistant" $\longrightarrow$ $\{K_A, K_B\}$ is cached, resolving "$K_{MIT}$ faculty assistant supervisor" can resolve "$K_A$ supervisor" and "$K_B$ supervisor" directly.

[3] Binary search is possible by fetching *T+2*, *T+4*, etc.

Figure 2 presents a name resolution algorithm that takes advantage of cached resolutions. Calls to **yield** return proofs of the resolution; calls to **insert** mark resolutions that are cached back into ConChord.

Another way to accelerate extended name resolutions is to leverage closure. For example, if we know that we will need to resolve the name "$K_{MIT}$ faculty assistants", we could create an entry whose *name* and *subject* are both that name and whose *sequence* is empty. We call such an entry a *truism*, as it simply states that a name is bound to itself. Since the subject of a truism is a name, it is stored in the `compatible` table. Closure then causes the values of the name to be stored in the `value` table, so the name can be resolved in a single lookup!

Li et al. [15] propose that membership checking can adapt between issuer-to-subject and subject-to-issuer searches to avoid large branching factors in the certificate graph. Implementing this algorithm on ConChord simply requires maintaining a subject-to-issuer table for entries and is an area of future work.

## 4 Evaluation

### 4.1 DNS Traces

We evaluate the effectiveness of name resolution sharing using a trace of 30,000 DNS requests captured at MIT's Laboratory for Computer Science [14]. We do not propose ConChord as a replacement for DNS; rather, we use the trace to generate a simple SDSI name hierarchy and a realistic name resolution workload. For each DNS address query of the form "`www.foo.com`", we generate a name resolution request "$K_{dns}$ com foo www" and a set of certificates:

$$\begin{aligned} K_{dns}\ \text{com} &\longrightarrow K_{dns.com} \\ K_{dns.com}\ \text{foo} &\longrightarrow K_{dns.com.foo} \\ K_{dns.com.foo}\ \text{www} &\longrightarrow K_{dns.com.foo.www} \end{aligned}$$

None of the certificates expire during the trace.

We run the trace between a single client and server and count the number of *sequential* lookups made for each request. While the total number of lookups for a name of length $l$ is $O(l^2)$ (due to fetching prefixes in parallel), the number of sequential lookups (thus, latency) is $O(l)$. Since every lookup is for a singleton set in the `value` table, we expect exactly $l$ sequential lookups per resolution.[4]

To evaluate the effectiveness of name resolution sharing, we run the trace with no caching, with caching of full name resolutions only (like a DNS proxy), and with caching of full names and name prefixes. Each resolution caches all its results (and prefixes) before the next one begins.

Figure 3 plots the cumulative distribution of sequential lookups per name resolution for each trace. With no caching, one lookup is made for each part of

[4] We could also have created truisms for each DNS hostname and thus reduced each name resolution to a single lookup. This might be reasonable, since each domain owner knows in advance what hostnames are valid and might be resolved.

```
resolve (name n)
    entry e
    e.name ← n
    e.subject ← n
    e.sequence ← ø
    resolve(e)

resolve (entry e)
    name n ← e.subject
    for i ← n.length down to 1
        set ← value[prefix(n,i)]
        for each e′ ∈ set
            entry r ← compose(e, e′)
            if (r.subject is a public key)
                yield(r)
                insert(r)
            else
                resolve(r)
                entry p ← extract(r)
                if(p ≠ e′)
                    insert(p)

// requires r.name = N · X and r.subject = K · X
// for some name N, public key K,
// and (possibly empty) sequence of strings X
// returns the extracted entry e
extract (entry r)
    e.name ← N
    e.subject ← K
    e.sequence ← r.sequence
    return e
```

**Fig. 2.** Name resolution

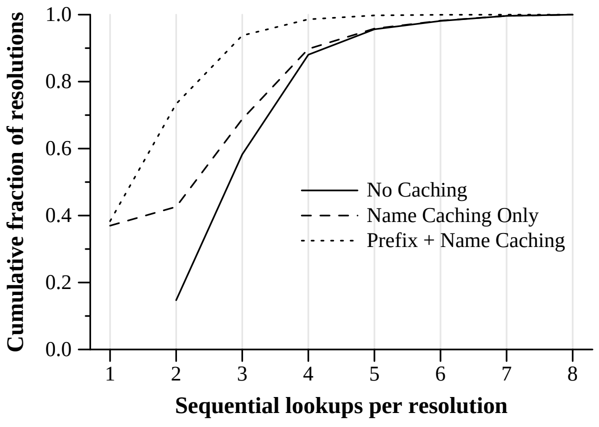

**Fig. 3.** Cumulative distribution of sequential lookups per DNS name resolution. The *No Caching* distribution is equivalent to the distribution of DNS name lengths.

the DNS name, so the distribution of lookups is the same as the distribution of name lengths. Full name caching reduces the mean by 23%, but variance is high, since many names are requested only once. Prefix caching reduces the mean by 43%, and 73% of the requests succeed in one or two lookups, suggesting that many requests share a common prefix.

We conclude that prefix caching is quite effective at reducing the latency of name resolutions for this dataset. This is not particularly surprising, as prefix caching is analogous to NS record caching in DNS (shown to be particularly effective in [14]). However, we believe that prefix caching will also benefit other datasets with hierarchical structures.

### 4.2 Mailing Lists

The DNS dataset is too simple to require closure over its certificates, so we evaluate the overhead of closure using a second dataset based on MIT course mailing lists. Course lists are composed of section lists, which are in turn composed of students, forming a widely-branching hierarchy of large groups. We gathered mailing lists for 27 courses, containing 38 sections and 2,073 students (1,706 distinct) and used the lists to generate a total of 5,624 certificates. For each entry of the form "`6.033-students: 6.033-sec9: alice`" (6.033 is a course number), we add the following certificates to an insertion trace (suppressing duplicates):

$$
\begin{array}{rlrl}
K_{mit}\ \text{registered} \longrightarrow & K_{mit}\ \text{courses students} & & \\
K_{mit}\ \text{students} \longrightarrow & K_{mit}\ \text{alice} & K_{mit}\ \text{alice} \longrightarrow & K_{alice} \\
K_{mit}\ \text{courses} \longrightarrow & K_{mit}\ \text{6.033} & K_{mit}\ \text{6.033} \longrightarrow & K_{6.033} \\
K_{6.033}\ \text{students} \longrightarrow & K_{6.033}\ \text{secs students} & K_{6.033}\ \text{secs} \longrightarrow & K_{6.033}\ \text{sec9} \\
K_{6.033}\ \text{sec9} \longrightarrow & K_{sec9} & K_{sec9}\ \text{students} \longrightarrow & K_{mit}\ \text{alice}
\end{array}
$$

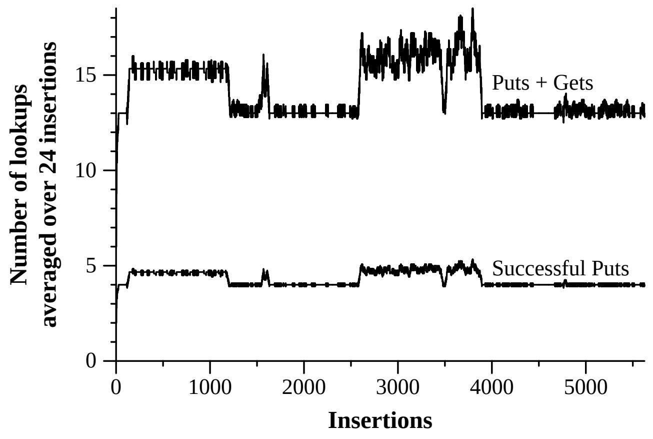

**Fig. 4.** Number of Chord lookups averaged over 24 insertions for the MIT course mailing list dataset. Each lookup is used either to "put" data in or "get" data from the system. Each insertion triggers a (nearly) constant number of other insertions to maintain closure. Insertions are grouped by course; the raised parts of the curves indicate additional compositions for insertions into courses with sections.

If the course does not have recitation sections, students are added directly to the course's "students" group. This dataset is designed to support a number of useful queries, such as determining whether Alice is registered in 6.033 or enumerating all the students registered in MIT courses.

We count the number of Chord lookups required to insert each certificate and the resulting derived certificates. Figure 4 shows that the number of lookups per insertion is fairly constant. This is because the number of compositions needed to maintain closure after adding a member to a group is proportional to the number of parent groups affected, which is usually small. For example, adding a student to a section only requires closure with the groups that transitively contain that section. The raised parts of the curves correspond to insertions for courses with sections, as these require one addition composition to maintain the section list. Reordering the trace changes the distribution of lookups per insertion, but does not affect the total number of lookups. We conclude that maintaining closure is practical for such datasets.

Since no access trace is available for this dataset, we cannot evaluate the total benefit that closure provides for name resolutions or membership checks. However, specific examples show that the benefit can be substantial: given closure, a membership check for any local name requires a single `check` lookup. Without closure, a check on a name like "$K_{MIT}$ registered" can require up to eight successive lookups to retrieve the necessary certificates.

## 5 Related Work

Alternatives to SDSI, such as DNSEXT [7] and X.509 [17], are used almost exclusively for Internet host identification, rather than applications like webs of trust or access control. While X.509 could support richer applications, it is not deployed in any way that facilitates them. PGP [24] supports user-authorized names and webs of trust, but not linked namespaces or named groups. PolicyMaker [3] and Keynote [2] support more general policies than SDSI, but they do not specify a way to locate the certificates needed to satisfy a particular policy.

Previous work [1, 15] proposes algorithms for resolving SDSI names using a distributed set of certificates, but does not address the practical challenges of storing and locating those certificates. Nikander and Viljanen [16] describe how to deploy SPKI/SDSI [21] using DNS, but do not support SDSI name resolution.

QCM [11] introduced *policy-directed certificate retrieval* as a general technique for locating the certificates needed to satisfy a given assertion. QCM and its successor, SD3 [13], use authoritative servers to implement distributed resolution of SDSI-like names and rely on embedded URIs or IPs to map principals to servers. ConChord supports policy-directed certificate retrieval to resolve SDSI names and eliminates the need for a mapping between principals and servers. While ConChord loses some of the benefits of authoritative servers, such as online signing and control over certificate dissemination, ConChord gains scalability, self-configuration, and load-balance.

## 6 Conclusion

We have presented ConChord, a distributed SDSI certificate directory built on a peer-to-peer system. ConChord supports three operations: membership checks, name resolutions, and certificate insertions. To accelerate checks and resolutions, ConChord maintains closure on each insertion and supports name resolution sharing. Experiments show that these techniques are effective and practical.

ConChord provides a novel deployment design that offers a number of practical advantages over traditional, hierarchical server architectures. ConChord eliminates any need to embed location information in certificates and automatically balances load among storage servers. Servers periodically reinsert entries to guarantee eventual consistency and can automatically resolve conflicts that occur due to network partitions.

Our prototype implementation supports the basic features described in this paper. Future work includes implementing replication, supporting SPKI/SDSI authorization certificates and revocation, limiting per-user storage, handling malicious failures, and generalizing ConChord for use with other certificate systems.

## Acknowledgments

We thank Barbara Liskov for her valuable advice on the presentation of this paper. We also thank Russ Cox, Robert Morris, Athicha Muthitacharoen, Ronald Rivest, Emil Sit, and the anonymous referees for their helpful comments.

## References

1. S. Ajmani. A Trusted Execution Platform for multiparty computation. Master's thesis, MIT, 2000. App A: Certificate Chain Algorithms.
2. M. Blaze, J. Feigenbaum, and A. D. Keromytis. Keynote: Trust management for public-key infrastructures (position paper). In *Security Protocols Workshop*, pages 59–63, 1998.
3. M. Blaze, J. Feigenbaum, and J. Lacy. Decentralized trust management. Technical Report 96-17, 28, 1996.
4. D. Clarke, J. Elien, C. Ellison, M. Fredette, A. Morcos, and R. L. Rivest. Certificate chain discovery in SPKI/SDSI. *Journal of Computer Security*, 2001.
5. R. Cox and A. Muthitacharoen. Serving DNS using Chord. In *Proc. IPTPS*, 2002.
6. F. Dabek, M. F. Kaashoek, D. Karger, R. Morris, and I. Stoica. Wide-area cooperative storage with CFS. In *Proc. ACM SOSP*, Oct. 2001.
7. DNS extensions (IETF DNSEXT), Mar. 1999. http://www.ietf.org/html.charters/dnsext-charter.html.
8. P. Druschel and A. Rowstron. PAST a large-scale, persistent peer-to-peer storage utility. In *HotOS VIII*, May 2001.
9. C. Ellison, B. Frantz, B. Lampson, R. Rivest, B. Thomas, and T. Ylonen. SPKI certificate theory. RFC 2693, Sept. 1999.
10. C. M. Ellison and D. E. Clarke. High speed TUPLE reduction. Memo, Intel, 1999.
11. C. A. Gunter and T. Jim. Policy-directed certificate retreival. Technical Report MS-CIS-99-07, U. Penn., Sept. 1998.
12. J. Y. Halpern and R. van der Meyden. A logic for SDSI's linked local name spaces. *Journal of Computer Security*, 9(1,2):47–74, 2000.
13. T. Jim. SD3: A trust management system with certified evaluation. In *Proc. 2001 IEEE Symposium on Security and Privacy*, May 2001.
14. J. Jung, E. Sit, H. Balakrishnan, and R. Morris. DNS performance and the effectiveness of caching. In *Proc. ACM SIGCOMM Internet Measurement Workshop*, 2001.
15. N. Li, W. H. Winsborough, and J. C. Mitchell. Distributed credential chain discovery in trust management. In *Proc. 8th ACM CCS*, Nov. 2001.
16. P. Nikander and L. Viljanen. Storing and retrieving internet certificates. In *Proc. 3rd Nordic Workshop on Secure IT Systems*, 1998.
17. Public-key infrastructure (IETF PKIX), Feb. 2000. http://www.ietf.org/html.charters/pkix-charter.html.
18. S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker. A scalable content-addressable network. In *Proc. ACM SIGCOMM*, 2001.
19. R. L. Rivest and B. Lampson. SDSI – A simple distributed security infrastructure. Apr. 1996.
20. A. Rowstron and P. Druschel. Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems. In *Proc. IFIP/ACM Middleware*, 2001.
21. Simple public key infrastructure (IETF SPKI), Feb. 1998. http://www.ietf.org/html.charters/spki-charter.html.
22. I. Stoica, R. Morris, D. Karger, M. Kaashoek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for Internet applications. In *Proc. ACM SIGCOMM*, Aug. 2001.
23. B. Y. Zhao, J. Kubiatowicz, and A. Joseph. Tapestry: An infrastructure for fault-tolerant wide-area location and routing. Technical Report UCB/CSD-01-1141, UC Berkeley, Apr. 2001.
24. P. R. Zimmermann. *The Official PGP User's Guide*. MIT Press, 1995.