ConChord: Cooperative SDSI Certificate Storage and Name Resolution

Sameer Ajmani, Dwaine E. Clarke, Chuang-Hue Moh, Steven Richman*
{ajmani,declarke,chmoh,richman}@lcs.mit.edu
MIT Laboratory for Computer Science

Abstract

SDSI is a proposed public key infrastructure that allows principals to define local names and link their namespaces to delegate trust. Unlike DNS, SDSI's egalitarian design resists deployment on traditional server hierarchies.

We present ConChord, a peer-to-peer deployment architecture for SDSI. ConChord provides load-balanced storage while eliminating many of the administrative difficulties of traditional architectures. ConChord introduces novel algorithms for resolving SDSI names in distributed systems and introduces new distributed data structures for storing SDSI certificates. Experiments show that maintaining our storage invariants is practical for large name hierarchies.

1 Introduction

SDSI (Simple Distributed Security Infrastructure) [13] is a proposed public key infrastructure that is more powerful and flexible than existing systems like DNSEXT [5] and X.509 [11]. 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 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 (such as embedding URIs in public keys, which seems undesirable). 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" [3]. Furthermore, providing fault-tolerance for these servers requires significant 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 [4, 6] configure themselves to provide immense storage capacity, high reliability, balanced load, and efficient lookups. ConChord uses the Chord [16] lookup system, with storage

*Authors in alphabetical order.

[1] Certificates on Chord

and caching techniques based on the Cooperative File System (CFS) [4]. ConChord locates certificates using relevant information (such as the name the 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 automatic conflict resolution. 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* [2]. A *reducing* certificate binds a local name to a principal. So, if Professor X wants to add Bob and Carol to her group of RAs, she can issue two reducing certificates:

$$\begin{aligned} K_{ProfX}\ \text{RAs} &\longrightarrow K_{Bob} \\ K_{ProfX}\ \text{RAs} &\longrightarrow K_{Carol} \end{aligned} \quad (1)$$

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_{ProfX} RAs" is the set $\{K_{Bob}, K_{Carol}\}$.

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

$$K_{MIT}\ \text{faculty} \longrightarrow K_{ProfX} \quad (2)$$

then we can resolve the extended name "K_{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 $\{K_{Bob}, K_{Carol}\}$. Bob can prove that he is a member of "K_{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:

$$\begin{aligned} K_{MIT}\ \text{staff} &\longrightarrow K_{MIT}\ \text{faculty} \\ K_{MIT}\ \text{staff} &\longrightarrow K_{MIT}\ \text{faculty RAs} \\ K_{MIT}\ \text{staff} &\longrightarrow K_{HR}\ \text{visiting} \end{aligned} \quad (3)$$

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_{MIT} staff" as the union of the values of "K_{MIT} faculty", "K_{MIT} faculty RAs", and "K_{HR} visiting".

Since the name bound by reducing certificate (2), "K_{MIT} faculty", is a prefix of the subject of non-reducing certificate (3), these certificates are called *compatible*, and we can *compose* (3)

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

Table 1: **ConChord Hash Tables**

with (2) to yield a new, derived certificate:

$$K_{MIT} \text{ staff} \longrightarrow K_{ProfX} \text{ RAs} \qquad (4)$$

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 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 [7]). 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 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 (we'll elaborate on this later).

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:

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

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

3.1 Peer-to-Peer Architecture

ConChord locates entries on servers using the Chord [16] 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.

A problem, however, arises with inserts. Concurrent inserts of two compatible certificates might fail to detect one another, so some valid compositions may be missed. The result would be a violation of the closure invariant upon which our algorithms depend. The closure invariant might also be violated if a client crashes before completing all derived insertions.

[2]ConChord could also have used the CAN [12], Pastry[14], or Tapestry[17] distributed lookup systems.

Rather than solve this problem with synchronization (which is slow in the wide area), each server periodically reinserts the check entries it stores to guarantee that all compositions eventually happen. This is an efficient solution because the work of reinsertion is spread among the servers, and reinsertions can be infrequent.

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.

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 certificates in the check table. If no new certificates are found, the server storing e informs the set size server. 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.

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 respon-

sible 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]).

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.

Li et al. [9] 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 would simply require maintaining a subject-to-issuer table for entries and is an area of future work.

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

4 Evaluation

We evaluate the overhead of maintaining closure using a 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 5,624 certificates. We count the number of Chord lookups required to insert each certificate and the resulting derived certificates.

The number of lookups to add members to a group is proportional to the number of parent groups affected, which is small. For example, adding a student to a section only requires closure of the course that contains that section. 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, we cannot evaluate the total benefit closure provides for resolutions or 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 in this dataset can require up to eight successive lookups.

Due to space constraints, we omit a set of tests using DNS traces. Results show that sharing name resolutions can effectively reduce name resolution latency for such datasets.

5 Related Work

Alternatives to SDSI, such as DNSEXT [5] and X.509 [11] lack the flexibility to support applications like webs of trust and access control and so are used almost exclusively for Internet host identification. PGP [18] supports user-authorized names but not linked namespaces.

Previous work [1, 9] proposes algorithms for resolving SDSI names using a distributed set of certificates, but does not address the practical challenges of storing and locating those certifi-

cates. Nikander and Viljanen [10] describe how to deploy SPKI/SDSI [15] using DNS, but do not support SDSI name resolution.

QCM [8] uses authoritative servers to implement distributed resolution of SDSI-like names, but QCM servers require complex manual configuration. In contrast, ConChord divorces security policies from the storage system, and so can use a self-configuring storage system.

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. Experimental results show that this is technique effective and practical.

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

The ConChord prototype supports the basic features described in this paper. Future work includes implementing replication, supporting SPKI/SDSI authorization certificates and revocation, and handling malicious failures.

Acknowledgments

We thank Barbara Liskov for her valuable advice on the presentation of this paper. We 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] 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.

[3] R. Cox and A. Muthitacharoen. Serving DNS using Chord. In *Proc. IPTPS*, 2002.

[4] F. Dabek, M. F. Kaashoek, D. Karger, R. Morris, and I. Stoica. Wide-area cooperative storage with CFS. In *Proc. ACM SOSP*, 2001.

[5] DNS extensions (IETF DNSEXT), 1999.

[6] P. Druschel and A. Rowstron. PAST a large-scale, persistent peer-to-peer storage utility. In *HotOS VIII*, 2001.

[7] C. M. Ellison and D. E. Clarke. High speed TUPLE reduction. Memo, Intel Corp., 1999.

[8] C. A. Gunter and T. Jim. Policy-directed certificate retreival. Technical Report MS-CIS-99-07, U. Penn., 1998.

[9] N. Li, W. H. Winsborough, and J. C. Mitchell. Distributed credential chain discovery in trust management. In *Proc. 8th ACM CCS*, 2001.

[10] P. Nikander and L. Viljanen. Storing and retrieving internet certificates. In *Proc. 3rd Nordic Workshop on Secure IT Systems*, 1998.

[11] Public-key infrastructure (IETF PKIX), 2000.

[12] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker. A scalable content-addressable network. In *Proc. ACM SIGCOMM*, 2001.

[13] R. L. Rivest and B. Lampson. SDSI – A simple distributed security infrastructure. 1996.

[14] 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.

[15] Simple public key infrastructure (IETF SPKI), 1998.

[16] 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*, 2001.

[17] 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, 2001.

[18] P. R. Zimmermann. *The Official PGP User's Guide*. MIT Press, 1995.