ConChord: Cooperative SDSI Certificate Storage and Name Resolution

Sameer Ajmani Dwaine Clarke Chuang-Hue Moh Steven Richman
{ajmani,declarke,chmoh,richman}@lcs.mit.edu

December 10, 2001

Abstract

SDSI is a secure, decentralized naming system that is well suited to access control and distributed group applications. However, SDSI's complex name resolution makes it challenging to deploy in distributed systems.

ConChord is the first system to support efficient and complete SDSI name resolution with a distributed certificate store. ConChord stores certificates using a peer-to-peer distributed hash table that provides load balance and fault-tolerance. ConChord accelerates name resolutions by maintaining closure over its certificates and allowing clients to share cached name resolutions.

ConChord is implemented using the Chord distributed lookup system and CFS-like replication and caching. ConChord distributes sets of certificates over many servers and supports weakly consistent updates to those sets.

Experimental results show that name caching accelerates real workloads by 40% and maintaining closure for a large group hierarchy requires reasonably low overhead per certificate insertion.

1 Introduction

SDSI [18] is a secure, decentralized naming system that is well suited to access control and distributed group applications. However, SDSI's complex name resolution makes it challenging to deploy in distributed systems.

ConChord[1] is the first system to support efficient and complete SDSI name resolution with a distributed certificate store. ConChord requires abundant, reliable storage with decentralized administration, and so uses the Chord [22] distributed lookup system, with replication and caching techniques based on the Cooperative File System (CFS) [6].

ConChord enables distributed clients to share secure name definitions for objects and groups. Clients can resolve names through ConChord, yielding authorization proofs that can be presented to access control systems or other secure applications. ConChord provides a unified cache for such proofs, allowing clients to leverage each other's resolutions.

ConChord's design goals and supporting techniques are:

Fast name resolution ConChord allows clients to cache name resolutions and use them in future resolutions.

Complete results On each certificate insertion, ConChord maintains closure over the certificate set (detailed in Section 3).

High availability ConChord replicates each certificate on several storage servers.

Load balance ConChord distributes sets of certificates among many storage servers and supports weakly consistent set updates.

ConChord's prototype implementation supports name resolution, closure maintenance, and basic certificate storage and retrieval, but not yet

[1] Certificates on Chord

replication. Experimental results show that name caching accelerates real workloads by 40% and maintaining closure for a large group hierarchy requires reasonably low overhead per insertion.

The rest of this paper is organized as follows: Section 2 discusses related work, and Section 3 provides background on SDSI. Section 4 presents ConChord's data structures, algorithms, and storage design. Section 5 evaluates the implementation, and Section 6 concludes.

2 Related Work

SDSI has been explored extensively by the theory community [1, 5, 13] focusing on algorithms for completeness and name resolution. SPKI/SDSI [21] is an IETF proposal that shares SDSI's naming scheme and defines formats for authorization certificates.

Previous work [2, 14] proposes algorithms for resolving SDSI-like names using a distributed certificate set, but does not address the practical challenges of storing and locating certificates in the wide area. These algorithms also fail to leverage the shared work of cooperating clients.

DNSSEC [7] and X.509 [16] provide secure name resolution in distributed systems, but require trusted roots and authoritative servers for their infrastructure. QCM [10] implements distributed resolution of SDSI-like names, but uses online servers to sign new certificates composed of older ones. ConChord does not require online private keys, since it stores certificate sequences explicitly. SFS [15] can implement SDSI-like naming using links between filesystems, but such names cannot identify groups of objects or individuals. All of these systems require explicit replication of authoritative servers to achieve fault-tolerance, whereas ConChord automatically replicates its data across servers.

3 SDSI Background

A SDSI *name* is a public key followed by one or more strings, such as "K_{Alice} friends" or "$K_{Teacher}$ students parents". Each SDSI name represents a set of public keys. A SDSI *certificate* is a statement that binds a name consisting of a single string to a *subject*; a subject can be a public key or a name. For example:

$$K_{Alice} \text{ friends} \longrightarrow K_{Bob} \tag{1}$$
$$K_{Alice} \text{ friends} \longrightarrow K_{Bob} \text{ parents} \tag{2}$$
$$K_{Bob} \text{ parents} \longrightarrow K_{Dad} \tag{3}$$

(1) is issued by K_{Alice}, Alice's public key, and binds the name "K_{Alice} friends" to K_{Bob}. To be considered authentic, (1) must be signed by Alice's private key and must be valid (an expiration date is embedded in the certificate). (2) binds "K_{Alice} friends" to the name "K_{Bob} parents", and (3) binds "K_{Bob} parents" to K_{Dad}.

SDSI certificates can be *composed* to create new certificates. For example, (3) can "rewrite" the subject of (2), so the *sequence* of certificates (2)(3) (with their signatures) represents the certificate:

$$K_{Alice} \text{ friends} \longrightarrow K_{Dad} \tag{4}$$

A sequence of certificates and signatures acts as a proof of a name-to-key binding. A sequence can, for example, be presented by a requestor to an access controller that restricts access to a resource to a named group. The controller can verify in linear time whether the sequence is valid and binds the group's name to the requestor's key.

Finding a sequence of certificates that prove a particular name-to-key binding requires searching through the set of certificates. Some certificate sets may cause a standard recursive search to be *incomplete*, i.e., loop infinitely or miss the results of some name resolutions.

Calculating the *closure* over the set of certificates ensures that resolutions are *complete* [5]. The closure contains all certificates formed by composing one certificate in the set with another, and is analogous to a transitive closure over a relation. ConChord maintains the closure invariant over its set of certificates on each certificate insertion.

4 Design

Each entry in ConChord contains a *name*, a *subject*, and a *sequence* that resolves that *name* to

Table	Index	Value
binds	name	set of entries whose *name* is name and whose *subject* is a public key
bound-to	name	set of entries whose *subject* is a name that starts with name
target	name, subject	an entry whose *name* is name and whose *subject* is subject

Table 1: ConChord Table Types

that *subject*. This entry format is self-certifying: clients can verify that an entry is correct without any additional information.

Like certificates, two entries can be *composed* to form a new entry; the composition has the *name* of the first entry, the *subject* of the first entry rewritten using the second entry, and the concatenation of the two entries' *sequences*.

Table 1 summarizes the entry tables used in ConChord. binds and bound-to contain sets of entries and are used to maintain closure and to resolve names. target supports fast group membership tests. These tables are implemented on a single distributed hash table.

4.1 Algorithms

Insertion and Closure Figure 1 presents **insert**, the algorithm to insert an entry e into ConChord. **insert** first checks whether e already exists in the target table, and, if so, returns. If not, **insert** adds e to the target table and continues.

If $e.subject$ is a public key, **insert** adds e to the set binds[$e.name$], fetches the entries in bound-to[$e.name$], and recurses (in parallel) on the compositions of those entries with e.

If instead $e.subject$ is a name, **insert** adds e to the set bound-to[*prefix-of(e.subject)*], fetches the entries in binds[*prefix-of(e.subject)*], and recurses (in parallel) on the compositions of e with those entries.

ConChord's insertion algorithm is based on ideas in [5] and [8]. We prove that the algorithm always terminates and is complete in our companion paper [4].

Lazy clients may not do the closure for entries they insert, but this can be repaired by periodically re-closing entries. Malicious clients can leverage closure to create many entries that use up storage space; repelling such attacks is an area of future work.

```
insert(entry e)
  if (target[e.name, e.subject] is empty)
    target[e.name, e.subject] ← e
    if (e.subject is a public key)
      binds[e.name] ← binds[e.name] ∪ {e}
      set ← bound-to[e.name]
      for each e′ ∈ set
        insert(compose(e′, e))
    else
      bound-to[prefix(e.subject, 1)]
        ← bound-to[prefix(e.subject, 1)] ∪ {e}
      set ← binds[prefix(e.subject, 1)]
      for each e′ ∈ set
        insert(compose(e, e′))

// returns the composed entry e
compose(entry e1, entry e2)
  e.name ← e1.name
  e.subject ← e2.subject · (e1.subject - e2.name)
  e.sequence ← e1.sequence · e2.sequence
  return e
```

Figure 1: Insertion with Incremental Closure

Name Resolution Figure 2 presents **resolve**, the algorithm to resolve a name n to the set of entries that bind n to public keys. First, **resolve**(n) creates a dummy entry e whose *name* and *subject* are n and whose *sequence* is empty.

Then, **resolve**(e) fetches the entries in binds for all prefixes of $e.subject$ (in parallel). **resolve** composes each of these entries with e to create a new entry r. If $r.subject$ is a key, **resolve** returns r and continues to the next composition; otherwise, **resolve** recurses on r.

After a **resolve** completes, it inserts all resolutions found for n and prefixes of n back into ConChord (these are shown by calls to **insert** in the figure; **extract** creates entries for prefixes). These cached results allow future resolutions to complete with fewer sequential lookups. For example, if "K_{dns} com foo" is cached, resolving "K_{dns} com foo www" can avoid intermediate

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

resolve (entry e)
   name n ← e.subject
   for i ← n.length down to 1
      set ← binds[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)

// returns the extracted entry e
extract (entry r)
   e.name.key ← r.name.key
   e.name.strings ← r.name.strings
                    - r.subject.strings
   e.subject ← r.subject.key
   e.sequence ← r.sequence
   return e
```

Figure 2: Name Resolution

steps.

Membership Checking To prove that a public key K is bound to name n, we look up target[n,K], which returns an entry for this binding if one exists. If this fails and n has length one (i.e. consists of a single string), ConChord's closure invariant guarantees that no such entry exists. If the target lookup fails and n's length is more than one, the client must resolve n using **resolve**.

Li et al. [14] propose that a membership check can adaptively choose between issuer-to-subject and subject-to-issuer searches to avoid large branching factors in the certificate graph. Implementing such an algorithm on ConChord is straightforward and is an area of future work.

4.2 Storage Architecture

ConChord's algorithms rely on an underlying distributed hash table that provides high capacity and availability, efficient storage and retrieval of large sets of entries, and decentralized administration. Peer-to-peer storage systems satisfy these needs.

Since ConChord entries are self-certifying, the servers providing storage need not be trusted to protect entry contents. However, our current design assumes servers obey our protocols; handling malicious failures is an area of future work.

ConChord's storage system is built using the Chord [22] distributed lookup system[2] with storage, replication, and caching modeled after CFS's DHash storage layer [6].

Chord allows N cooperating servers to organize themselves so that locating the value associated with a *Chord ID* requires at most $log(N)$ hops between the servers. Servers may join and leave the system at any time; Chord lookups are robust to network and node failures.

DHash is a strategy for replicating and caching data items on Chord to provide load balance and fault tolerance. DHash replicates a data item on several servers that can be efficiently located when looking up the item. DHash caches items along each clients' lookup path; other clients are likely to find cached entries since their lookups tend to converge to the same set of servers.

Storing Entries and Sets Each index into ConChord's hash tables is mapped to a Chord ID by hashing the index with the table's name. Two of the tables, binds and bound-to, store sets of entries that need to support updates by multiple users.

ConChord's entry sets require very weak consistency: sets are essentially append only, because every participant has the same notion of deletion (certificate expiration), and the ordering of elements in a set is unimportant. Recent systems have shown the benefits that can be reaped from varying the level of update isolation and consistency to suit application needs

[2] CAN [17], Pastry[19], or Tapestry[23] could also be used.

[20, 9]. ConChord leverages this idea.

We choose to graft support for updates onto DHash rather than use update support from OceanStore [12]. OceanStore requires manual configuration of replica groups and enforces a total order on updates. In contrast, DHash selects replica groups automatically. Also, ConChord does not require the strength of OceanStore's update model.

Given ConChord's lax consistency requirements, updatable sets can be implemented by modifying DHash as follows: add a new atomic *append-to-set* operation to storage servers, and store each set on the node responsible for the set's Chord ID (and that node's replicas). Replicas can apply updates in any order they like, with the guarantee of eventual consistency.

Unfortunately, this simple scheme scales poorly to large sets. A very large set might cause load imbalance or even exceed the capacity of a single node. Further, large sets are difficult to cache. Cache inconsistencies (caused by new set insertions) are difficult to handle inexpensively. Whole-set invalidations and expirations would lead to the re-copying of large amounts of data, so a more complicated hierarchical update propagation scheme would probably be necessary.

Distributing Sets ConChord avoids these problems by distributing entries in a set among several nodes. The members of the set whose Chord ID is s are stored at Chord IDs $hash(s \oplus 1) \ldots hash(s \oplus T)$, where T is the size of the set. The value of T is stored as a *set size record* at ID s. Nodes support three atomic operations on set size records: *get-size*, *increment-and-get*, and *decrement*.

To fetch the members of a set, a client calls *get-size* to get the set's 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 a set with ID 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 Chord ID $hash(s \oplus T')$.

When an entry in a set expires, the node storing that entry can compact the set. The node fetches the last element in the set from node n, stores it locally, tells n to free the space (n verifies that no elements are stored later in the set, so as to not create holes), and decrements the set size record.

Consistency Issues All update operations use primary copy replication [3]. In the absence of network failures, this guarantees that each location in a set contains exactly one entry. However, because Chord/DHash lacks a well defined notion of replica group view changes, network failures can cause different clients and servers to have inconsistent views of replica groups. We propose (but have not proven correct) the following protocol to handle transient network failures.

Clients and servers can detect and recover from incorrect set size records by fetching past the expected end of the set until no more entries are found (location $T+1$, $T+2$, etc.[3]). When a client or server detects that a set is larger than expected, it reports back to the node responsible for the set size record.

Different replicas in the same group might accidentally store different entries during a network partition. When the partition heals, replicas temporarily store all entries accepted by any replica in the group (at most one entry per replica). The group then lazily agrees to divert all but one entry to the end of the set (content hashes provide a total order to determine which entry is not diverted).

We have not yet implemented this replication protocol; implementing and evaluating it is an area of future work.

To balance load among servers, set size records and set entries can be cached along lookup paths as in DHash. However, they may become out-of-date (DHash avoids this problem with content hashes). Therefore, we assign time-to-live values to cache entries. Cached set size records need to have relatively small TTLs (but recall that out-of-date size values can be tolerated by peeking

[3]Binary search is possible by fetching $T+2$, $T+4$, $T+8$, etc.

Case	insert	resolve
worst	$O(ln^3)$	$O(l^2n^2)$
average	$O(lb^l)$	$O(l^2b^l)$
trivial	$O(l)$	$O(l^2)$

Table 2: Algorithm Complexity Bounds on table operations: n is the number of certificates, l is the length of a name, and b is the number of keys bound to a name (the branching factor).

a bit past the expected end of the set). Cached set entries can have relatively large TTLs, since the only time cached entries can become invalid is when a replica group diverts an entry.

5 Evaluation

We derive complexity bounds for **insert** and **resolve** in [4] and present them in Table 2. The worst case occurs with a pathological certificate set that is extremely unlikely to appear in practice. The remaining two cases are considered in our experiments.

Our experiments measure the number of *table operations* (fetches and inserts) made by ConChord's algorithms. Each target table operation requires one Chord lookup, or about 430ms [6]. Operations on binds or bound-to require two sequential Chord lookups (one for the set size and one for the entries), except for singleton sets that require only one.

5.1 DNS Trace

We evaluate ConChord's name resolution performance and caching effectiveness using a trace of 30,000 DNS requests captured at MIT's Lab for Computer Science [11]. 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.

Each DNS name "`www.foo.com`", e.g., is used to generate a name resolution request "K_{dns} com foo www" and is mapped to a set of certificates:

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

This dataset corresponds to the *trivial* case in Table 2, because the branching factor (b) is always one. Name lengths (l) vary from two to eight. Each certificate insertion requires exactly three table operations, since no name needs closure.

We run the trace between a single client and server and count the number of *sequential* table operations made for each request. While the total number of operations is $O(l^2)$, the number of sequential operations (thus, latency) is $O(l)$. To evaluate the effectiveness of caching, we run the trace with no caching, with caching of full names only, and with name and prefix caching.

Figure 3 plots the operations per request averaged over the trace. The cumulative standard deviation for each curve is fairly constant and is given on the plot. Figure 4 plots the distribution of requests that require various numbers of operations.

With no caching, one query is made for each part of the DNS name, so its distribution is the distribution of name lengths. Full name caching reduces the mean by 23%, but variance is high. Prefix caching reduces the mean by 43%, and 73% of the requests succeed in one or two queries. Each query requires one Chord lookup, since all sets are singletons.

Prefix caching provides substantial benefit for this dataset, as it is analogous to NS record caching in DNS [11]. We believe prefix caching will also benefit other datasets with hierarchical structures.

5.2 Mailing Lists

The DNS hierarchy is too simple to exercise closure on insertion, so we present a second set of tests using MIT course mailing lists. Course lists are composed of section lists, forming a widely-branching hierarchy of large groups. We do not evaluate name resolutions on this dataset because no access trace is available.

We gathered the mailing lists for 27 MIT courses, containing 38 sections and 2073 students (1,706 distinct). We map these lists to a SDSI group hierarchy that includes parent

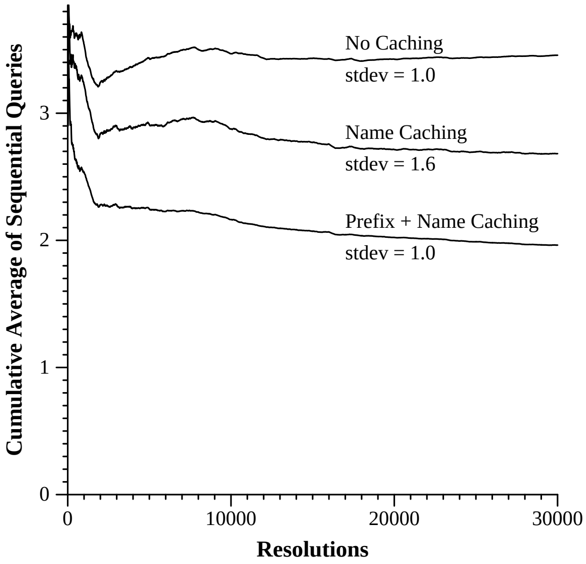

Figure 3: Cumulative average of sequential queries needed to resolve DNS names. Cumulative standard deviation for each curve is fairly constant over the trace.

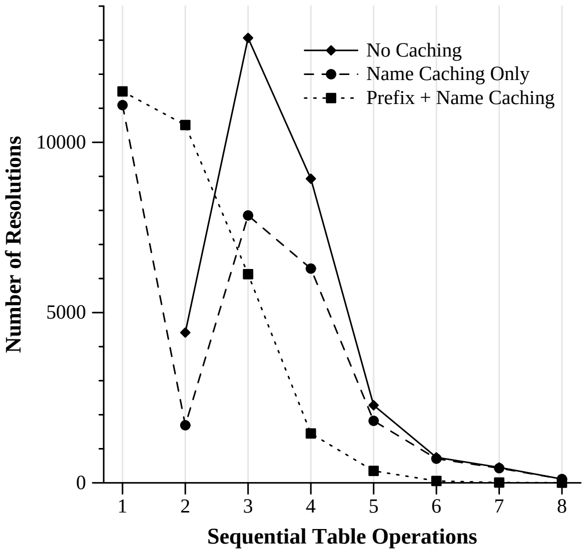

Figure 4: Distribution of sequential queries per DNS name resolution.

groups that require closure over all the students. For each entry of the form "`6.033-students: 6.033-sec9: alice`", we add the following certificates, in order, to an insertion trace (suppressing duplicates):

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

If the course does not have recitation sections, students are added directly to the course's "students" group. Finally, we prepend the certificate "K_{mit} registered $\longrightarrow$ K_{mit} courses students" to the beginning of the trace, so it is updated on each insertion.

This dataset corresponds to the *average* case in Table 2. Name lengths (l) vary from one to two, and branching factors (b) vary from one to 1706. The number of operations per insertion can never exceed the number of students.

Figure 5 plots the number of table operations averaged over 24 insertions in the trace. The upper curve shows the total number of operations per insertion ($\mu = 6.5$, $\sigma = 3.0$, min 3, max 12). The lower curve shows the number of entries successfully added per insertion ($\mu = 4.4$, $\sigma = 2.0$, min 2, max 8). The raised parts of the curves correspond to insertions for courses with sections, since one additional step is required per insertion to maintain closure.

The number of operations to add members to a group is proportional to the number of parent groups affected, which is small. For example, adding a student only requires closure with the groups that transitively contain that student. Most of these operations require two Chord lookups, but some (operations on the `target` table) require only one. Many of the operations for each insertion occur in parallel.

This result is partially due to the order of insertion: if the "registered" certificate were inserted last, it would require an operation for ev-

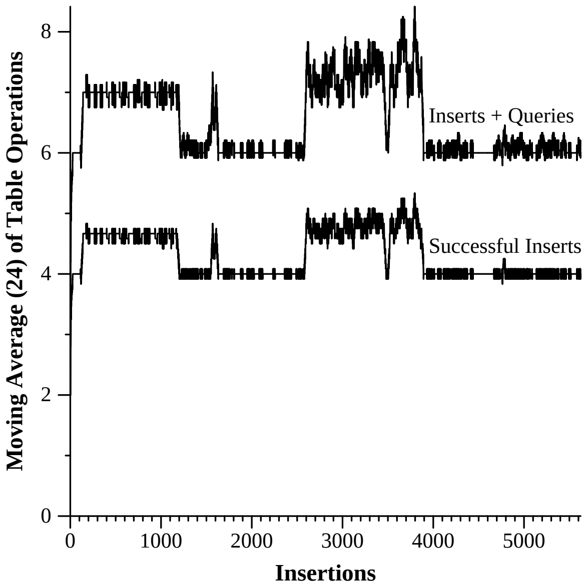

Figure 5: Number of table operations averaged over 24 insertions for the MIT course mailing list hierarchy. Each insertion triggers a constant number of other insertions to maintain closure. Raised parts are for courses with sections.

ery student in every course. However, the number of operations done for each student would decrease, so the total number of operations required to insert this certificate set would stay the same. We conclude that the amortized overhead of maintaining closure on each insertion is low.

6 Conclusion

ConChord is a large-scale, reliable certificate store that supports efficient SDSI name resolution. ConChord accelerates and guarantees completeness for name resolutions by maintaining closure and caching previous resolutions. Experimental results show that these techniques are effective and require low overhead.

ConChord supports a simple update model for sets of entries that balances load among storage servers. Set sizes are stored at servers and can be cached to reduce load on those servers. Clients can detect and recover from out-of-date set sizes.

We have implemented a prototype of ConChord that supports the basic features described in this paper. Areas of future work include implementing replication, handling malicious failures, reducing storage requirements, and evaluating ConChord in the wide area with richer datasets.

References

[1] M. Abadi. On SDSI's linked local name spaces. *Journal of Computer Security*, Oct. 1998.

[2] S. Ajmani. A Trusted Execution Platform for multiparty computation. Master's thesis, MIT, July 2000.

[3] P. Alsberg and J. Day. A principle for resilient sharing of distributed resources. In *Proc. 2nd Intl. Conf. on Software Engineering*, Oct. 1976.

[4] D. Clarke, S. Ajmani, C.-H. Moh, and S. Richman. Correctness and complexity of ConChord, 2001. Draft.

[5] 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. To appear.

[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] Domain name system security (dnssec), Mar. 1999.

[8] C. M. Ellison and D. E. Clarke. High speed TUPLE reduction. Technical report, Intel Corp., 1999. Personal Memo.

[9] A. Fox, S. D. Gribble, Y. Chawathe, E. A. Brewer, and P. Gauthier. Cluster-based scalable network services. In *Proc. ACM SOSP*, Oct. 1999.

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

[11] J. Jung, E. Sit, H. Balakrishnan, and R. Morris. DNS performance and the effectiveness of caching. In *Proc. ACM SIGCOMM Internet Measurement Workshop*, 2001.

[12] J. Kubiatowicz et al. OceanStore: An architecture for global-scale persistent storage. In *Proc. ACM ASPLOS*, Nov. 2000.

[13] N. Li. Local names in SPKI/SDSI. In *Proc. 13th IEEE CSFW*, pages 2–15, July 2000.

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

[15] D. Maziéres, M. Kaminsky, M. F. Kaashoek, and E. Witchel. Separating key management from file system security. In *Proc. ACM SOSP*, Dec. 1999.

[16] Public-key infrastructure (X.509) (PKIX), Feb. 2000.

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

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

[19] 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.

[20] Y. Saito, B. N. Bershad, and H. M. Levy. Manageability, availability, and performance in Porcupine: a highly scalable, cluster-based mail service. In *Proc. ACM SOSP*, Dec. 1999.

[21] Simple public key infrastructure (SPKI), Feb. 1998.

[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.