Secure Distributed DNS

Christian Cachin
IBM Research
Zurich Research Laboratory
CH-8803 Rüschlikon, Switzerland

Asad Samar [†]
Deptartment of ECE
Carnegie Mellon University
Pittsburgh PA-15217, USA

Abstract

A correctly working Domain Name System (DNS) is essential for the Internet. Due to its significance and because of deficiencies in its current design, the DNS is vulnerable to a wide range of attacks. This paper presents the design and implementation of a secure distributed name service on the level of a DNS zone. Our service is able to provide fault tolerance and security even in the presence of a fraction of corrupted name servers, avoiding any single point of failure. It further solves the problem of storing zone secrets online without leaking them to a corrupted server, while still supporting secure dynamic updates. Our service uses state-machine replication and threshold cryptography. We present results from experiments performed using a prototype implementation on the Internet in realistic setups. The results show that our design achieves the required assurances while servicing the most frequent requests in reasonable time.

1. Introduction

The Domain Name System (DNS) is one of the most critical parts of the Internet infrastructure and maps symbolic *domain names* to IP *addresses*. A *name server* that fails may deliver incorrect name-to-address mappings to its clients and can cause services to become unreachable or, even worse, to be masqueraded by fraudulent replacements operated by an attacker.

The domain name space has the structure of a tree in which every node has a label. The *domain name* of a node is the list of the labels on the path from the node to the root of the tree. A *domain* is identified by a domain name, and consists of that part of the domain name space that is at or below the domain name which specifies the domain. A domain is a subdomain of another domain if it is contained within that domain [13]. The DNS tree is divided into *zones*. A zone is a contiguous portion of the tree (possibly containing many domains) that is treated as a unit for administrative purposes [10].

For every zone there are multiple *authoritative* name servers which manage the corresponding part of the name space. Only they know all data for the zone, including where to find the servers with the authority for the delegated subdomains. When a client wants to obtain data on a domain name, it usually invokes a close-by name server who retrieves the data in a series of queries to authoritative servers along the path from the root node to the target name.

The authoritative servers of every zone are replicated to protect against failures. They are usually divided into a *primary* and one or more *secondary* servers. As the names imply, the original zone data is kept at the primary server and the secondary servers periodically obtain it from the primary (or from another secondary) using a master/slave approach. This means that an attacker may corrupt the data of all servers by compromising the primary alone.

The DNS Security Extensions (DNSSEC) [2] [9] have been introduced to protect the DNS; their most important technique is *zone signing*, which adds digital signatures to zone data, computed by the entity who owns the zone. However, zone signing has not been widely used so far. One reason for this is that the private signature key of the zone must be guarded very tightly and is, therefore, often stored offline (and nobody has yet taken on the responsibility for the root key!). At the same time, zone data is updated ever more frequently using Dynamic DNS updates [17], requiring an online signing process for signed zones.

This work presents a solution for the above problems by using secure replication for authoritative servers and by distributing the signature keys so that they can be kept online. Our solution addresses the security of zone replication and enables secure dynamic updates — two problems which have recently been identified to be among the most important open problems in DNS security [3]. In particular, our method prevents an attacker from spoofing an entire zone by compromising its primary server and ensures that no name server becomes a single point of failure.

† Work done at IBM Zurich Research Laboratory.

Our approach can tolerate one third of the authoritative name servers for a zone to fail in arbitrary and malicious ways, while maintaining the correct behavior of the name service as a whole. This allows a client to obtain correct and consistent responses from the zone's name service even though some of the name servers have been compromised by an attacker. We configure the authoritative name servers of a zone as replicated state machines [15] and use a fault-tolerant atomic broadcast protocol to disseminate the state updates. The implementation of atomic broadcast is provided by the SINTRA prototype [6]. It tolerates a fraction of the servers to behave arbitrarily and exhibit so-called "Byzantine faults." SINTRA uses an asynchronous network model, which is appropriate for the Internet.

We further address the issue of where to store the private signature keys of a signed zone; we keep them online, as required for dynamic updates, but in a distributed way through threshold cryptography [8]. Thus, even when a fraction of the servers are corrupted, the secret is still not revealed. We use the non-interactive RSA threshold signature scheme of Shoup [16], which produces standard RSA/SHA-1 signatures that can be verified by DNSSEC clients.

Our prototype service implements the existing interface as defined by DNSSEC and hence no current DNSSEC client needs to be modified in order to communicate with it. The system does affect the performance of the DNS operations, however, and increases the latencies seen by the clients.

We have carried out experiments with our system to assess its performance. The setup used replicated name servers in Europe and in North America connected by the Internet, which is a typical DNS setup for a multinational corporation. Our benchmarks show that retrieval and update operations can still be serviced in a very reasonable time, even when the servers are not in close proximity of each other.

We review the background required to understand the essentials of our system in Section 2. The design is presented in Section 3. We give some implementation details in Section 4 and discuss the performance results in Section 5. Section 6 concludes the paper.

2. Background and Related Work

Vulnerabilities in the DNS have frequently been exploited for attacks on the Internet. One of the most common ways of "defacing" a web server is to redirect its domain name to the address of a host controlled by the attacker through manipulation of the DNS. DNSSEC [9] eliminates some of these problems by providing end-to-end authenticity and data integrity through *transaction signatures* and *zone signing*.

Transaction signatures are computed by clients and servers over requests and responses. DNSSEC allows the two parties either to use a message authentication code (MAC) with a shared secret key or public-key signatures for authenticating and authorizing DNS messages between them. The usefulness of transaction signatures is limited since they guarantee integrity only if a client engages in a transaction with the server who is authoritative for the returned data, but do not protect against a corrupted server acting as a resolver.

For *zone signing*, a public-key for a digital signature scheme, called a *zone key*, is associated with every zone. Every *resource record* (it is the basic data unit in the DNS database) is complemented with an additional SIG resource record containing a digital signature, computed over the resource record.[1] Zone signing also protects relayed data because the signature is created by the entity who owns the zone.

We denote the private signature key of a zone by sk_{zone} and the corresponding public key by pk_{zone}. Clients are assumed to know pk_{zone} — either obtained from the DNS, signed under the key of the parent zone, or statically configured into the client — and can verify the SIG records on all data retrieved from the server. This guarantees that responses are authentic. But unless further transport-level security methods are used, only transaction signatures guarantee the *freshness* of retrieved data.

During dynamic updates [17], the primary must modify some resource records. With zone signing, the server has to sign the modified resource records using sk_{zone}. This means that the key must be kept online, which substantially increases the risk that it is compromised.

Threshold cryptography [8] is a well-known technique to eliminate the danger that the private key of a cryptosystem is exposed at a single location. An (n,t)-*threshold signature scheme* is a protocol that allows a set of n severs to share the private key of a signature scheme and to issue signatures collaboratively. Any set of $t+1$ servers may generate a signature, but any set of t or fewer servers does not obtain information about the private key. Threshold implementations are known for many digital signature schemes; we use the the RSA scheme described by Shoup [16], which is the first practical and provably secure scheme of its kind and, moreover, produces standard RSA signatures.

The idea of using a threshold signature scheme for zone signing has previously been explored by Wang *et al.* [18]. They provided benchmarks for signature operations on replicated servers. In another related work, Wu, Malkin, and Boneh [19] realized an intrusion-tolerant secure web server running SSL and a certification author-

1 Actually, SIG records span a set of resource records, but we ignore this distinction here.

ity using threshold cryptography. However, in both of these cases, all servers were connected to the same LAN and no fault-tolerant broadcast protocols were used.

Threshold cryptography is only one half of our approach. The other half is to use *state-machine replication* [15] with *atomic broadcast* for disseminating the requests. An atomic broadcast protocol [11] guarantees that every replicated server receives and executes the same sequence of requests, even though the servers only communicate over point-to-point links and some of them may be faulty. Since the DNS operations are deterministic, this ensures consistency among the non-faulty replicated servers.

For our Internet setting, we use *asynchronous* atomic broadcast with “Byzantine” faults, which can cope with arbitrary message delays, needs no synchronized clocks, and tolerates the corrupted servers to behave arbitrarily. Such protocols have only recently been developed; examples are the BFT system by Castro and Liskov [7], the protocols by Cachin et al. [4] and by Kursawe and Shoup [12] implemented in SINTRA [6], and the COCA system by Zhou, Schneider, and van Renesse [20].[2]

In another related work, Ahmed [1] employs techniques detailed in [7] to achieve fault tolerance for DNS. However, the issue of storing zone keys online is not addressed in detail. Furthermore, the underlying protocols of [7] are deterministic and have to make certain timing assumptions to ensure liveness in the face of byzantine faults..

3. Design

3.1. System Model

A *request* in our system represents a DNS request that can either be a query or an update. A query request, e.g., to obtain the IP address corresponding to a domain name, is called a *read request*. An update request, e.g, to add a new name-address mapping to the zone data, is called a *write request*. We use *server* to denote a name server that services a request (read or write) and *client* to denote a name server or a resolver that issues a request.

Our replicated name service for a particular zone consists of the static set of n authoritative servers (also called *replicas*), who interact with multiple clients that send read and write requests to the servers. Any $t < n/3$ of the servers may fail arbitrarily due to a configuration error or a malicious action of an adversary. The faulty servers are called *corrupted* and may also collude with the adversary. All other servers are called *honest*. We assume that initially all honest servers are in the same state, i.e., they have the same zone data.

2 Although COCA does not implement atomic broadcast, it would also be suitable for zone signing.

For defining our goals in a precise way, we introduce a single abstract entity called a *trusted server* that always follows the specification. The trusted server is initialized in the same state as the honest servers of our service and interacts with multiple clients. A write request from a client causes the trusted server to update its state. We assume w.l.o.g. that each write request is followed by a request that reads the complete state of the server. We say a response from our service is *correct* if it is the same as the response generated by the trusted server when it interacts with the same clients as the service.

Furthermore, we assume every client that makes a request has a method to verify the response received from the service (such a response may consist of multiple messages). We say the response is *acceptable* if this verification is successful.

We assume that each pair of servers is connected through a point-to-point authenticated reliable link, i.e., the origin of each message sent can be authenticated and each message sent is eventually received by the peer. There are no bounds on the message delay on a link and no common clocks.

3.2. Goals

As our goal is to build a name service that behaves correctly even though t servers may be corrupted, no action must rely on one server alone that might become a single point of failure.

If secure DNS is used, it also means that the zone key must be stored in such a way that it is not leaked to the corrupted servers, but still allows for dynamic updates. This implies that the service is capable to use the zone key for issuing digital signatures online.

We require our replicated name service to satisfy the following goals in the presence of up to t corrupted servers:

G1. (**correctness**) Every acceptable response is correct.

G2. (**liveness**) If a client sends a request to the service, then an acceptable response is eventually returned to the client.

G3. (**secrecy**) The zone key is stored and used (for dynamic updates) in such a way that no information about it is leaked to the corrupted servers.

These goals can be made formal using the simulatability paradigm of modern cryptography.

The actual implementation should support the interfaces specified by the DNS and DNSSEC standards such that any secure DNS client may use the service without changes. Furthermore, we want to use as much of the existing DNS infrastructure and code as possible.

3.3. System Design

To achieve the required fault tolerance against corrupted servers, we integrate the existing replication in the DNS with state-machine replication [15]. Our design here assumes that links between clients and servers are authenticated, e.g., using transaction signatures.

The replicated service performs three basic steps: It receives a request from the client, processes the request, possibly moving to a new state, and returns a response. We describe these three steps in detail below.

Disseminating Requests. Recall that only the primary server in DNS executes dynamic updates and propagates changes to secondary servers using a master/slave approach. We configure all replicas in primary mode so that each replica maintains its own master copy of the zone data. Clients send their requests to all replicas, who use an atomic broadcast protocol to agree on a sequence of requests. We use the "optimistic" asynchronous atomic broadcast protocol of Kursawe and Shoup [12] implemented in SINTRA; it requires $n > 3t$, which is optimal in the given model. Optimistic atomic broadcast works in two modes: a fast optimistic mode where a leader orders the requests, and a slower fall-back mode, where the replicas proceed using a distributed protocol. The protocol switches to the fall-back mode when the leader is apparently not performing correctly, in the view of the other replicas, and invokes a Byzantine agreement protocol [5] to establish a new leader and a consistent state across the group.

To perform dynamic updates, each replica then executes the requests as a primary and commits any changes to its local zone data. Any write request must be authorized by a transaction signature of the client.

Since read requests do not change the state of a server, one might think that they do not need to be disseminated using atomic broadcast. However, our approach requires that all honest servers also send the same response back to the client (so that the client can determine which one is correct by majority vote). For the honest replicas to generate the same response, they must agree on the order in which read requests are processed with respect to the write requests. Therefore, we also disseminate the read requests using atomic broadcast.

Processing Requests. Any read request or a write request in an unsigned zone is processed trivially according to the DNS specifications by each replica once the request is delivered through atomic broadcast. A write request in a DNSSEC-signed zone, however, requires computing SIG records on the modified data using the zone key sk_{zone}, which is shared using an (n, t)-threshold signature scheme. When processing a write request in a signed zone, the servers run the following *threshold signature protocol* to compute the signature for a new or modified SIG record.

The server generates its signature share using the share generation algorithm of the threshold scheme that takes as input the data to be signed and the server's share of the zone key sk_{zone}. The resulting signature share is then sent to all servers in a point-to-point message, and the server waits until it receives valid signature shares from $t + 1$ distinct servers (the server can determine the validity of a signature share by running a local share verification algorithm). Finally, the server invokes the signature assembly routine of the threshold signature scheme that computes a valid threshold signature from $t + 1$ valid shares.

Since $t < n/2$, this mechanism ensures that the replicated service can perform dynamic updates without leaking information about the zone's private key to the corrupted servers, thus achieving G3.

Sending Responses. Because up to t replicas might be corrupted, G1 requires the client to accept a response only if this response represents at least $t + 1$ replicas. In our design, every replica sends a response message directly to the client. The client receives $n-t$ responses from distinct replicas, determines the majority value among them, and accepts this value as the response of the service.

The accepted value is correct (G1) because at most t servers are corrupted and every honest server sends the same reply in state-machine replication. Thus, the $n - t \geq 2t + 1$ received messages include the responses of at least $t+1$ honest replicas. Liveness (G2) is satisfied for the same reasons since each message sent by an honest replica eventually reaches the client.

3.4. A Pragmatic Approach

Although the approach described above achieves our goals, it requires modifying all DNS client applications — something that we would rather avoid. Existing clients send a DNS request only to a single server (instead of sending it to all replicas using atomic broadcast) and do not support any mechanisms for obtaining multiple responses and combining them in a majority vote.

A scheme for secure service replication where the clients must only minimally be changed has been proposed by Reiter and Birman [14]. Following their approach, we could still achieve our goals without modifying the clients by using transaction signatures with a threshold scheme and sharing the private key among the servers. Responses would be signed using the threshold signature and verified by existing DNSSEC clients. However, computing a threshold signature on every response is a very expensive operation, as our results in Section 5 will show. The costs associated with this approach would be prohibitive because a DNS server predominantly handles read requests, which should not involve a signing operation.

Therefore, we propose a pragmatic approach, which works without modifying existing client applications, but achieves only slightly weaker goals. In particular, we have to relax G1 and G2.

Achieving G1 is hard for an unmodified client since a client waits only for a single response message from a name server. Although with zone signing, a corrupted replica may not send an acceptable response that it has made up at will, it could send data that is no longer valid because this data has been updated in the mean time (akin to a *replay attack*). It is easy to see that this attack can only be prevented by either contacting multiple servers (which would require changing the client) or by using a threshold transaction signature (which would not be practical).

Achieving G2 is also not possible unless the client contacts multiple servers since a request sent to a single corrupted server might simply be ignored and no response will ever be returned. Therefore, the client has to contact at least one honest server to guarantee liveness.

Hence, we weaken the model as follows. Consider a *weak trusted server* that has the same initial state as the honest replicas in our service and interacts with multiple clients. When receiving a read request, the weak trusted server may return the correct value (according to its specification) or it may behave as if it had received the request at any previous point in time during its execution and return the resulting value. Moreover, the weak trusted server may ignore any request (actually, this matters only for write requests). We then define a response from our service to a particular client request to be *approximate* if it is the same as the response generated by the weak trusted server when it interacts with the same clients as the service.

In our pragmatic design, we now require the replicated name service to satisfy the following goals in the presence of up to t corrupted servers:

G1$'$. (**weak correctness**) Every acceptable response is approximate.

G2$'$. (**weak liveness**) If a client sends a request to an honest server, then an acceptable response is eventually returned to the client.

The protocol that implements our pragmatic design for signed zones is the same as before, with zone signing using threshold signatures, except for the following two changes: First, the client sends a request only to a single replica. This server acts as a gateway and disseminates the request to other replicas using atomic broadcast. The second modification is that the client does not perform a majority vote and receives only one response.

Note that the client will still receive multiple response messages, but depending on the implementation, it will choose either the message from the gateway or the message that arrives first. The response is accepted only if it contains the proper signatures under the zone key.

This scheme achieves G1$'$ since the DNS data in each response is signed using the zone key, which requires at least $t+1$ signature shares, so an acceptable response must have been signed by honest servers at some point in time (but the response is not necessarily fresh and may not reflect all write requests sent by the clients). G2$'$ is achieved because the atomic broadcast protocol will eventually deliver every message sent by an honest replica. Note that the scheme also achieves G3.

In practice, our pragmatic approach gives a stronger liveness guarantee than G2$'$ because of the existing replication in the DNS and because most existing DNS clients make timing assumptions. When a DNS client does not receive a response for some request after a timeout period, it resends the request to another authoritative server for the domain. With the typical round-robin scheduling used by most clients, this guarantees liveness in a partially synchronous model.

This design allows a modified client (according to Section 3.3) to actually achieve G1 and G2 since it sends requests to and combines responses from multiple replicas. Our mechanism, therefore, is incrementally deployable: unchanged clients can use it to obtain the weaker G1$'$ and G2$'$, whereas modified clients also achieve G1 and G2.

In zones where updates occur only rarely, the read operations do not even have to be disseminated using atomic broadcast and occur at no additional cost compared to unmodified secure DNS with zone signing.

3.5. Optimizations

As shown in Section 5, the most expensive operation in the system is the computation of the threshold signatures. We describe two modifications of the basic threshold signature protocol from Section 3.3 to make it more efficient.

The main drawback of the RSA-based threshold signature scheme is that it is slow even when there are no corrupted servers. The signature shares in our scheme [16] consist of a *share value*, which is needed to assemble the final signature, and a *(correctness) proof*, which is a non-interactive zero-knowledge proof of knowledge that allows a receiver to verify the correctness of the share value. Generating and verifying such a proof is more expensive than generating the share value or assembling the final signature. These operations are also far more expensive than verifying the final signature, which is almost for free when a small public exponent is used.

Since all that matters is a correct final signature, the idea is to eliminate the proof from the critical path in the likely case that all servers are honest. We explore two ways to achieve this in the following "optimistic" threshold signature protocols.

*Optimistic Signature Protocol with Proofs (*OPTPROOF*).* In our first optimization, the proofs are generated and verified only on demand. Each server proceeds as follows. It generates a share value without a correctness proof and sends this to all servers. The server then receives $t+1$ shares without verifying their correctness, assembles them to a putative signature, and verifies only that. If it is valid, it must be the correct signature. In this case, the server sends the final signature to all other replicas and returns it to the application.

In the other case (if the signature was invalid), the server sends a request message to all other servers asking them to compute the signature shares again, but this time along with the proofs. Upon receiving such a request, each server generates the proof for its signature share and sends the share along with this proof to all servers. These shares with proofs are then processed in the same way as in the unoptimized algorithm. In parallel to the last two steps, the server also waits for receiving a valid final signature and terminates as soon as a correct signature is received.

Note that the step of waiting in parallel for a valid signature is necessary since it is not guaranteed that there are still enough honest replicas around to resend their shares with proofs — they might already have terminated the protocol. But then they have sent out the correct signature and it will be received by the waiting server.

*Optimistic Signature Protocol with Trial and Error (*OPTTE*).* Protocol OPTPROOF above works well when all servers are honest, but its performance is worse than the unoptimized protocol in the presence of corrupted servers. We therefore propose another optimization that performs much better in the presence of corrupted servers than both schemes above, but works only for relatively small n. For practical values of n, however, it turns out to be the fastest variation.

This protocol exploits the fact that there is only a limited number of corrupted servers, and so if a server collects enough shares from distinct servers in a set, then this set will contain a subset consisting of enough correct shares. This subset is found by trial and error.

The protocol starts out like our first optimistic protocol, where no server computes a proof. Every server sends only its share to all others. A server then receives $t+1$ shares and tries to assemble them to a valid signature. If it fails, it continues to receive signature shares and tries to assemble every subset of $t+1$ shares to a final signature, until at most $2t+1$ shares have been received. This is guaranteed to succeed because there are at most t invalid shares. However, the algorithm may take exponential time in n when t is a fraction of n.

4. Implementation

We have implemented the pragmatic approach from Section 3.4, relying on the SINTRA prototype for atomic broadcast and RSA threshold signatures. Our implementation further includes the basic threshold signature protocol from Section 3.3 and the two optimized versions described in Section 3.5.

4.1. Structure

The prototype uses BIND, the Berkeley Internet Name Daemon, which is the most widely used DNS server implementation. The servers run a modified version of named from BIND snapshot 9.3.0s2002111, and the clients in our system use the standard dig program (for read requests) and the nsupdate program (for write requests). We used this snapshot of the development version of BIND instead of the most recent release (BIND 9.2.2) because this was the only version available that had support for RSA signatures with SHA1 message digest.

The SINTRA prototype is implemented in Java, whereas BIND is written in C. We link them using a module called Wrapper implemented in Java, which is the main component of our replicated name service implementation.

Wrapper acts as a proxy between the clients and the original named from BIND. Wrapper runs on every authoritative name server of the zone and intercepts all requests from clients on UDP port 53. It then interacts with SINTRA for atomic broadcast and threshold signatures, and with named for DNS operations. Wrapper reads its configuration parameters from a file. These parameters include the values of n and t, the identities of all servers for the zone, and the threshold signature protocol to use (c.f., Section 3). Then it starts the SINTRA runtime system and opens an atomic broadcast channel among all members of the group.

Internally, Wrapper runs several threads. These threads include two dispatcher threads that listen for client requests and for requests from named for computing threshold signatures, respectively. Upon receiving such a request, these threads dispatch the request to one of several worker threads. The Wrapper also has a LinkReader and a ChannelReader thread, which interact with the communication primitives provided by SINTRA. LinkReader listens for asynchronous point-to-point messages from Wrapper modules running on other servers, and ChannelReader listens for messages that are sent through atomic broadcast.

4.2. Operation

Every server runs Wrapper and a modified copy of named, which is configured to listen on a different port than 53.

When Wrapper receives a client request to the name server on port 53, it broadcasts the request on the atomic broadcast channel to all servers in the group. Any such request delivered by atomic broadcast and received by ChannelReader is forwarded to named on the respective port. When named has finished processing the request, it sends the answer back to Wrapper, which returns it directly to the client. All communication between named and Wrapper is done through datagram sockets such that named believes it is communicating with a client. Note that Wrapper does not have to interpret client requests since its operation is independent of the particular request; every request is simply forwarded on the atomic broadcast channel to all servers.

When named performs a dynamic update in a signed zone data, it must compute new SIG records on the modified data. The signature routine of named has been modified so that it forwards the request on another datagram socket to a dispatcher thread in the local Wrapper. The request is then processed using one of the threshold signature protocols from Section 3. The output of this protocol is returned by Wrapper to named, and named completes the request as usual.

4.3. Initialization

SINTRA requires manual key distribution before it can be invoked. In particular, there is a key generation utility that must be run by a trusted entity and that outputs the private key shares of every server for several threshold public-key encryption and signature schemes used by SINTRA's protocols. The file with these private keys must be transported over a secure channel to every server (typically using SSH).

Creating a DNSSEC signed zone requires creating a zone key and signing the zone data using this key. BIND provides a utility for this. For ease of implementation, we have so far not generated the threshold key shares from BIND's zone key or modified this utility to generate threshold key shares, but use a threshold signature key generated by the initialization utility of SINTRA. The corresponding public key is then included in the zone data and the key shares are included in the private data of every server, which is distributed analogously to the initialization data for SINTRA.

When the replicated name service first begins operating, we assume that all servers store the same zone file. A special command may then be invoked on a single server to sign the zone data using the distributed key.

4.4. Details

For the purpose of testing the signature generation protocol, we can also configure a server to misbehave and to mimic a corrupted server. A server that is corrupted in this way inverts all the bits in its signature share before sending it to the others

We note that although the design of our system does not impose any synchrony assumptions on the distributed system model, the implementation is not entirely asynchronous. Specifically, the communication in the current SINTRA prototype implementation is based on TCP, which involves timeouts. Furthermore, the client applications dig and nsupdate use a timeout to decide how long to wait for a server response and will contact the next server in the list if the timeout expires (c.f., Section 3.4).

5. Benchmarks

5.1. Setup

Our experimental setup has two parts, one on a local-area network and the other one on the Internet over large distances. The local setup consists of a cluster of four machines at the IBM Zurich Research Laboratory connected by a 100 Mbits/s switched Ethernet. Experiments on the local setup provide a base case to determine how much of the latency in the Internet setup is caused by the network and how much by the protocol itself.

The Internet setup includes the cluster of four machines in Zurich and three more machines: one at the IBM T.J. Watson Research Center in New York, one at the IBM Austin Research Laboratory and one at the IBM Almaden Research Center in San Jose, connected by the IBM intranet. Table 1 shows the operating system, CPU type, clock speed, and the Java VM version of each machine. The machines are standard x86-architecture laptops and desktops running Linux. They represent the best choices we could obtain with reasonable effort and hence the set is rather diverse; for example, the machines in Austin and Almaden are considerably faster than those in Zurich and New York.

Our Internet setup mimics a typical DNS installation used by large international organizations, where a zone corresponding to a geographic site like Zurich is served from a small local cluster of name servers, close to where most of the queries arise, together with a few name servers at remote locations that serve as backups. Figure 1 shows the setup along with the average round-trip latencies measured on each link.

Location	# of machines	OS	CPU	MHz	Java
Zurich	4	Linux 2.2.x	P II	266	IBM 1.4.1
New York	1	Linux 2.2.x	P II	300	IBM 1.4.1
Austin	1	Linux 2.4.x	dual P III	1260	Sun 1.4.2
San Jose	1	Linux 2.4.x	P III	930	Sun 1.4.2

Table 1. Details of machines used in the experiments.

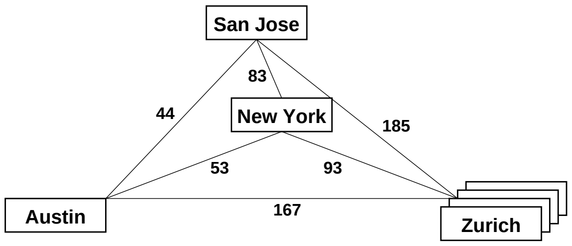

Figure 1. The experimental setup on the Internet with average round-trip times in milliseconds.

In the experiments that involve corrupted servers, we configure one of the servers in the Zurich cluster to be corrupted when there is only a single corruption; with two corrupted servers, we configure the Zurich server and the server at Austin to simulate corrupted behavior.

The prototype consists of the Java modules mentioned in Section 4, the modified named from BIND, and the SINTRA prototype written in Java. The RSA public-key signatures are provided by SINTRA, where they are implemented using the standard Java BigInteger package. Zone signatures use 1024-bit RSA moduli with SHA-1 and PKCS #1 encoding.

5.2. Experiments

We measure the time taken by the replicated name service to carry out a read, add, or delete request issued from dig or nsupdate, and to send back a response to the client. The requests are always sent by a client on the Zurich LAN to one of the servers in the Zurich cluster. We repeat these experiments for several values of n and t and with the three different threshold signature protocol variations presented in Section 3.

Table 2 shows the results (with times in seconds). Each number in the table is the average of 20 measurements run in sequence. The first column gives the setup using a pair of values (n, k), where n is the number of servers, $t = \frac{n-1}{3}$ corrupted servers can be tolerated, and k servers are actually simulating corrupted behavior.

The first row denoted by $(1, 0)$ represents the performance of the base case using the unmodified name server (from the used version of BIND) installed on one of the Zurich machines. Other than this base case, three different groups of machines are considered: the $(4, 0)*$ row, corresponds to the four identical machines on the local-area network in Zurich. The next two rows with a group of four replicas each correspond to a setup with two machines in Zurich, one in New York and another one in San Jose. The rows with seven servers include all servers.

The column marked "Read" shows the time taken by a read request to be returned to the client; recall that this involves the request to be sent over atomic broadcast (except for the base case $(1, 0)$ with the unmodified named). Results for read operations are shown only for the cases where no server is corrupted since a simulated corrupted server would not have an influence on the result. The case $(4, 0)*$ on the LAN represents the time taken by the atomic broadcast protocol itself since the link latencies are negligible. In the other cases, the time taken for a read can mostly be attributed to the latencies of the network links connecting the servers.

The other columns show the time taken by write requests, in particular for adding a record to the database and deleting a record from it. For both operations, measurements are given for each one of the three threshold signature protocols discussed in Section 3: the unoptimized protocol (denoted by BASIC), the optimistic protocol OPT-PROOF with correctness proofs, and the optimistic protocol OPTTE with trial and error.

In nsupdate, each add or delete request to an authoritative server is preceded by a read request. Thus, the data shown here includes the time taken for the read in addition to the subsequent add or delete operation. The numbers show that an add request takes almost twice as much time to process as a delete request in all cases. This is because the time is dominated by the computation of the threshold signatures and because named sequentially computes four new SIG records with signatures for an add request and two for a delete request.

5.3. Discussion

The results show that read operations take anywhere from around 50 milliseconds on the LAN to several hundred milliseconds, when remote machines on the Internet are involved. Write operations take from one to more than 20 seconds, depending on the topology, the scheme used, and the attacker's behavior.

(n,k)	Read	Add			Delete		
		BASIC	OPTPROOF	OPTTE	BASIC	OPTPROOF	OPTTE
$(1,0)$	0.003	0.047			0.022		
$(4,0)*$	0.05	7.09	1.72	1.53	3.80	0.96	0.92
$(4,0)$	0.37	6.36	3.09	3.01	3.10	1.78	1.80
$(4,1)$		9.29	6.48	3.10	5.04	3.99	1.90
$(7,0)$	0.44	21.73	3.06	2.30	10.09	1.74	1.83
$(7,1)$		24.57	4.20	3.46	10.85	2.73	2.03
$(7,2)$		21.21	15.79	4.01	10.55	8.32	2.27

Table 2. Experimental results with times taken by every operation in seconds.

Comparing the threshold signature protocols (BASIC, OPTPROOF, and OPTTE), we observe that the BASIC variant performs poorly even when there are no corrupted servers. The BASIC signature protocol spends most of the time for threshold signature generation and verification.

The fact that with the BASIC signature protocol, the $(4,0)*$ setup with all machines on the LAN is slower than $(4,0)$ with the machines distributed over the Internet can be attributed to the compute-bound nature of the algorithm and the unequal processing power of the involved machines; the four machines on the LAN are very slow compared to the ones in Austin and San Jose.

We observe that the optimized signature protocols decrease the time taken by write requests by a factor of four to six, to around 3 seconds for an add and 1–2 seconds for a delete operation over the Internet (in the $(4,0)$ and $(7,0)$ cases). An interesting feature apparent from these results is that the performance of the OPTPROOF protocol deteriorates much faster with an increasing number of corrupted servers than that of the OPTTE protocol; in particular, consider the $(7,2)$ case, where OPTPROOF takes almost as long as BASIC, but OPTTE is still a factor of 4–5 faster.

Table 3 shows a breakdown of the time taken to compute one threshold signature in the $(4,0)*$ case on the LAN using the BASIC protocol (recall that the threshold signature protocol is executed four times for add requests and twice for deletes). More than 96% of the time is spent for share generation and share verification, of which most can be attributed to generating and verifying the correctness proofs. Assembling and verifying the signature is negligible in comparison.

These results show that our secure replicated name service adds a small delay to read requests and services write requests in a reasonable time. Write requests take more time because they involve threshold signature computations, but updates are typically much less frequent than reads.

6. Conclusions

Our replicated name service provides fault tolerance and security guarantees to secure DNS against an attacker that compromises a fraction of name servers in a zone, while supporting dynamic updates. Since the approach requires $n > 3t$ servers for tolerating the corruption of t servers, small organizations running only two or three replicas today would need to deploy additional servers in order to benefit from our replication technique.

The results show that such a system can be used in practice. In particular, for highly critical parts of the DNS, like root servers or other servers near the root, our service can provide increased security. Although dynamic updates have high latency in our current implementation, one should remember that they are not a frequent operation compared to reads and, moreover, occur less often when one gets closer to the root of the name space.

References

[1] S. Ahmed, *A Scalable Byzantine Fault Tolerant Secure Domain Name System.* Master's thesis, MIT, Jan. 2001.

[2] R. Arends, R. Austein, M. Larson, D. Massey, and S. Rose, "Dns security introduction and requirements," `draft-ietf-dnsext-dnssec-intro-09.txt`, Feb. 2004.

[3] D. Atkins and R. Austein, "Threat analysis of the domain name system," `draft-ietf-dnsext-dns-threats-03.txt`, June 2003.

[4] C. Cachin, K. Kursawe, F. Petzold, and V. Shoup, "Secure and effi cient asynchronous broadcast protocols (extended abstract)," in *Advances in Cryptology: CRYPTO 2001,* (J. Kilian, ed.), pp. 524–541, Springer, 2001.

[5] C. Cachin, K. Kursawe, and V. Shoup, "Random oracles in Constantinople: Practical asynchronous Byzantine agreement using cryptography," in *Proc. 19th ACM Symposium on Principles of Distributed Computing (PODC),* pp. 123–132, 2000.

[6] C. Cachin and J. A. Poritz, "Secure intrusion-tolerant replication on the Internet," in *Proc. Intl. Conference on Dependable Systems and Networks (DSN-2002)*, pp. 167–176, June 2002.

[7] M. Castro and B. Liskov, "Practical Byzantine fault tolerance and proactive recovery," *ACM Trans. Computer Systems*, vol. 20, pp. 398–461, Nov. 2002.

	generate share	verify share	assemble sig.	verify sig.
absolute [s]	0.82	0.78	0.05	0.003
relative [%]	49.6	47.2	3.0	0.2

Table 3. Breakdown of the time taken in the BASIC threshold signature protocol (message delays are negligible because the experiment was performed on the local setup).

[8] Y. Desmedt, "Threshold cryptography," *European Transactions on Telecommunications*, vol. 5, no. 4, pp. 449–457, 1994.

[9] D. E. Eastlake, "Domain name system security extensions," RFC 2535, March 1999.

[10] R. Elz and R. Bush, "Clarifications to the DNS specification," RFC 2181, July 1997.

[11] V. Hadzilacos and S. Toueg, "Fault-tolerant broadcasts and related problems," in *Distributed Systems*, (S. J. Mullender, ed.), New York: ACM Press & Addison-Wesley, 1993. Expanded version appears as Technical Report TR94-1425, Department of Computer Science, Cornell University, Ithaca NY, 1994.

[12] K. Kursawe and V. Shoup, "Optimistic asynchronous atomic broadcast," Cryptology ePrint Archive, Report 2001/022, March 2001. `http://eprint.iacr.org/`.

[13] P. Mockapetris, "Domain names - concepts and facilities," RFC 1034, Nov. 1987.

[14] M. K. Reiter and K. P. Birman, "How to securely replicate services," *ACM Transactions on Programming Languages and Systems*, vol. 16, pp. 986–1009, May 1994.

[15] F. B. Schneider, "Implementing fault-tolerant services using the state machine approach: A tutorial," *ACM Computing Surveys*, vol. 22, pp. 299–319, Dec. 1990.

[16] V. Shoup, "Practical threshold signatures," in *Advances in Cryptology: EUROCRYPT 2000*, (B. Preneel, ed.), pp. 207–220, Springer, 2000.

[17] P. Vixie, S. Thomson, Y. Rekhter, and J. Bound, "Dynamic updates in the domain name system," RFC 2136, March 1999.

[18] X. Wang, Y. Huang, Y. Desmedt, and D. Rine, "Enabling secure on-line DNS dynamic update," in *Proc. 16th Annual Computer Security Applications Conference (ACSAC'00)*, 2000.

[19] T. Wu, M. Malkin, and D. Boneh, "Building intrusion-tolerant applications," in *Proc. 8th USENIX Security Symposium*, pp. 79–91, 1999.

[20] L. Zhou, F. B. Schneider, and R. van Renesse, "COCA: A secure distributed online certification authority," *ACM Trans. Computer Systems*, vol. 20, no. 4, pp. 329–368, 2002.