A Scalable Byzantine Fault Tolerant Secure Domain Name System

by

Sarah Ahmed

Submitted to the Department of Electrical Engineering and Computer Science
In partial fulfillment of the requirements for the degree of

Master of Engineering in Computer Science and Engineering

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

January 22, 2001
[February 2001]

Author .... [illegible]
Department of Electrical Engineering and Computer Science
January 22, 2001

Certified by .... [illegible]
Barbara Liskov
Ford Professor of Engineering
Thesis Supervisor

Accepted by .... [illegible]
Arthur C. Smith
Chairman, Department Committee on Graduate Theses

A Scalable Byzantine Fault Tolerant Secure Domain Name System

hy

Sarah Ahmed
Massachusettes Institute of Technology

Submitted to the
Department of Electrical Engineering and Computer Science

On January 22, 2001

In Partial Fulfillment of the Requirements for the Degree of
Master of Engineering in Computer Science and Engineering

Abstract

The domain name system is the standard mechanism on the Internet to advertise and access important information about hosts. At its inception, DNS was not designed to be a secure protocol. The biggest security hole in DNS is the lack of support for data integrity authentication, source authentication, and authorization. To make DNS more robust, a security extension of the domain name system (DNSSEC) was proposed by the Internet Engineering task force (IETF) in late 1997. The basic idea of the DNS security extension is to provide data integrity and origin authentication by means of cryptographic digital signatures. However, the proposed extension suffers from some security flaws.

In this thesis, we discuss the security problems of DNS and its security extension. As a solution, we present the design and implementation of a Byzantine-fault-tolerant domain name system. The system consists of 3f+1 tightly coupled name servers and guarantees safety and liveness properties assuming no more than f replicas are faulty within a small window of vulnerability. To authenticate communication between a client and a server to provide per-query data authentication, we propose to use symmetric key cryptography. To address scalability concerns, we propose a hierarchical organization of name servers with a hybrid of iterative and recursive query resolution approaches. The issue of cache inconsistency is addressed by designing a hierarchical cache with an invalidation protocol using leases. Because of the use of hierarchical state partitioning and caching to achieve scalability in DNS, we develop an efficient protocol that allows replicas in a group to request operations from another group using very few messages. We show that the scalable Byzantine-fault-tolerant domain name system, while providing a much higher degree of security and reliability, performs as well or even better than an implementation of the DNS security extension.

Thesis Supervisor: Barbara H. Liskov
Title: Ford Professor of Engineering

Acknowledgements

I would like to thank my advisor, Barbara Liskov, for her constant support, encouragement, and advice to accomplish this work. Her careful and critical comments significantly improved the content and presentation of this thesis. She taught me what it means to be dedicated to research. She is, and will always be a source of inspiration to me to pursue the frontiers of computer science.

Many thanks to Miguel Castro whose work on a practical Byzantine fault tolerance inspired this thesis. His insight, guidance, and help were much valuable. I am especially thankful because he patiently answered my questions and provided me with deep insights even when he was busy trying to finish his own thesis.

I would also like to thank Zheng Yang who originated the work on a Byzantine-fault-tolerant DNS. His code was the basis of my implementation, which saved a lot of time. And special thanks to ChonChon and Chandra who provided me with much help while expanding my horizon with their diverse level of interests.

I would like to thank all my lab partners who made it a great pleasure to be a graduate student in the Programming Methodology group. This includes Sameer Ajmani, Chandrasekhar Boyapoti, Miguel Castro, Kincade Dunn, Kyle Jamieson, Rodrigo Rodrigues, Ziqiang Tang, and Yan Zhang.

This list would be incomplete without mentioning the contribution of my wonderful friends Imran and Bobby, who have tolerated my eccentricities with much patience and met all my demands, no matter how unreasonable. I am lucky to have friends like them.

I do not know how to thank my parents and siblings enough for their constant level of support and inspiration. Their support is what keeps me going to pursue my goals in life. Thanks mom and dad for being there for me.

And finally, I would like to thank my husband, Zahid Hasan, without whose love, understanding and support it may not have been possible to achieve what I have today. Life without him for the last five years was painful. We spent thousands of hours on the phone while he was at Stanford and me at MIT. But the simple feeling that he was there for me when I needed him filled my heart with great constancy, and inspired me to keep going. Thanks Zahid for being the wonderful person you are. I can't wait to see you soon.

Contents

1. Introduction **9**

1.1 Our Contribution 10

1.1 What the System Offers 11

1.2 Thesis Outline 12

2. DNS Infrastructure **13**

2.1 Name Space 13

2.2 Administration 14

2.3 Name Servers and Resolvers 15

2.4 Query Resolution 17

2.5 Resource Records 18

3. DNS Security Issues **23**

3.1 Problems with current DNS 23

3.1.1 Lack of Authentication 23

3.1.2 Lack of Consistency Control 24

3.1.3 Vulnerability to Server Failure 25

3.2 Proposed DNS Security Extension (DNSSEC) 25

3.3 Problems with DNSSEC 29

3.3.1 Zone Private Key Storage 30

3.3.2 Freshness Attack due to Pre-generated SIG RR 31

3.3.3 Lack of Consistency Control and Single Point of Failure 31

3.4 Why Need Something More Secure than DNSSEC 32

4. A Practical Byzantine-fault-tolerant Algorithm **33**

4.1 The CLBFT Algorithm 33

4.2 The System Model and Assumptions of the CLBFT 33

4.3 How the CLBFT Algorithm Works 34

4.3.1 View Changes 35

4.3.2 Proactive Recovery 36

4.4 The Properties of the CLBFT Algorithm 37

5. A Byzantine-fault-tolerant DNS with Session Key **39**

5.1 Design Criteria 39

5.2 The SBFTDNS System Architecture 41

5.3 Key Management 42

5.3.1 A Hybrid Resolution with Intermediate Level Hierarchy ... 48

5.3.2 Replicated Clients 49

5.4 Caching Mechanism 51

5.4.1 Security and Cache Consistency 51

5.4.2 A Hierarchical Invalidation Protocol using leases 53
5.4.3 Summary of Caching....................... 56
5.5 Running Operations.. 57

6. Implementation **59**
6.1 The CLBFT Replication Library 59
6.2 TIS/DNSSEC 60
6.3 SBFTDNS with Session Key Mechanism.................................... 61
6.4 Modifications to Provide System Scalability 63
6.4.1 Replicated Clients 63
6.4.2 RR Caching 63
6.4.3 Lease Mechanism 65
6.4.4 Invalidation 66
6.4.5 Dynamic Update 67
6.4.6 Reducing Number of Messages across Levels of Hierarchy.. 67
6.5 Related Work 69

7. Performance Evaluation **71**
7.1 Experimental Setup 71
7.2 Experiments 71
7.3 Results 73
7.3.1 Read-only Operations 73
7.3.2 Read-write Operations 75
7.3.3 System Performance Comparison 77
7.4 Performance Issues 78
7.4.1 Session Key Management 78
7.4.2 Replicated Clients and RR Caching 79
7.4.3 Reducing the Number of Messages Across Levels 79
7.5 Discussion 81

8. Conclusions **83**
8.1 Summary 83
8.2 Future Work 85
8.2.1 Improving the Implementation 85
8.2.2 Use of a Standard DNSSEC Package 85
8.2.3 Backward Compatibility 86
8.2.4 Realistic Performance Evaluation 86

A. An Analysis of DNS Traffic Patterns and Performance **89**
A.1 Motivation 89
A.2 DNS traces 90
A.3 Analysis 90

A.3.1 General Characteristics 90
A.3.2 Request latency 91
A.3.3 Number of Referrals 92
A.3.4 Query Type 92
A.3.5 Response Error 92
A.3.6 Root Server Characteristics 92
A.3.7 Caching NS Records 95
A.3.8 Effect of Aggregation on Cache Hit Rate 95
A.4 Discussion 96

References **98**

List of Figures

2-1 DNS Name Space Hierarchy 14
2-2 Distinction between a domain and a zone 15
2-3 A Typical DNS Query Resolution 18
3-1 DNS attack by breaking into a router 23
3-2 Man-in-the-middle attack of DNS by network packet spoofing 24
3-3 DNS attack by issuing fraudulent update request 24
3-4 DNS attack by a Single Name Server Break In 25
3-5 A Sample DNS Query Resolution in DNSSEC 28
4-1 Normal case operation in CLBFT. Replica 0 is the primary, replica 3 is faulty 35
4-2 View Change Protocol 36
5-1 A Fully Iterative Resolution Scheme 44
5-2 A Hybrid of Iterative and Recursive Resolution Schemes 45
5-3 A Fully Recursive Resolution Scheme 46
5-4 A Fully Recursive Resolution Scheme with Intermediate Hops 47
5-5 A Hybrid Resolution Scheme with n=2 Intermediate Levels of Hierarchy 48
5-6 Increase in the Number of Messages Across Levels of Hierarchy 50
5-7 A Scheme to Support Replicated Clients with Reduced Number of Messages 50
5-8 TTL and Object Staleness 52
5-9 A Hybrid of Client-driven and Server-driven Invalidation schemes with Leases 55
5-10 Advantage of Hierarchical Caching 56
5-11 Hierarchical Organization of Name Servers 57
6-1 The CLBFT Replication Library API 59
6-2 Architecture of TIS/DNSSEC 61
6-3 Architecture of SBFTDNS 62
6-4 Hierarchical Organization of Name Servers 63
6-5 A Scheme to Support Replicated Clients with Reduced Number of Messages 67
7-1 Response Latency for A Record 77
7-2 Response Latency for NS Record 77
7-3 Total number of Messages Traversing through the Network 80
7-4 Total number of Reply Bytes Traversing through the Network 80
A-1 PDF of DNS request latency 91
A-2 CDF of DNS request latency 91
A-3 CDF of the number of referrals 92
A-4 Latency distribution vs. number of referrals 92
A-5 Latency distribution across query types 93
A-6 Latency distribution across roots 94
A-7 Latency when root servers return answers vs. referrals 94
A-8 Cache Hit and Miss 95
A-9 Effect of Aggregation on Cache hit Rate 96

Chapter 1

Introduction

The Domain Name System (DNS) [25] is a distributed database coordinated by the DNS protocol to provide information crucial to the operation of the Internet. DNS is the standard mechanism on the Internet to advertise and access important information about hosts. Virtually all internetworking services, including the World Wide Web, electronic mail, remote terminal access, and the file transfer protocol use DNS.

Because of its critical role in the Internet infrastructure, it is important that DNS tolerate failures and attacks both at the servers and during data communication over the Internet. These failures can either be due to benign faults, e.g., a server crash, or due to malicious attacks by hackers. However, DNS originally was not designed to be a secure protocol. The biggest security hole in DNS is the lack of data authentication. A resolver has no way to verify the authenticity and integrity of the data returned by the name servers. This means an attacker can easily fake the IP address of the sender included in DNS response packet, and the client has no choice but to trust the fake address as the origin of the reply data.

Also, DNS uses a simple primary-secondary scheme to ensure service availability and server load distribution. However, there is no consistency mechanism between the primary and the secondary name servers of DNS. During updates, data only gets updated in the database of the primary name server and the change is later propagated to the secondary name server during zone transfer. This means the secondary name servers in a highly dynamic zone almost always contain some stale data, which is inconsistent with the data provided by the primary name server. This causes consistency and reliability problems since clients may get stale data from some servers without an intentional attack. Moreover, some newly updated data may be lost from the system forever if the primary name server crashes before the data is transferred to the other servers. DNS thus does not tolerate even benign server faults.

Moreover, DNS makes extensive use of resource record caching. However, caching raises concerns about cache inconsistency and staleness of data. The original design of the domain name system sacrificed consistency in favor of reduced access time. For a secure domain name system, stale information may no longer be considered harmless, since it may involve some security critical information, e.g., a compromised private key. However, the current DNS protocol does not support any means to propagate data updates or invalidations to DNS servers in a fast and secure way.

Furthermore, only one name server is contacted at a time to provide name service in current DNS. This means one compromised server is enough to fool a client with wrong information, or corrupt the database by issuing fraudulent update requests, and there is no mechanism in DNS to overcome this vulnerability to server failure due to both benign faults and malicious attacks.

To solve the security problems of current DNS, a security extension [14] has been proposed by the Internet Engineering Task Force (IETF). The main idea of the extension (DNSSEC) is to provide data integrity and authentication using pre-generated digital signatures for each data item stored in the database. The name servers return the response along with its digital signature so that the client can verify the authenticity and integrity of the reply data.

While the DNS security extension provides data integrity and source authentication, it has some security holes too. DNSSEC ensures secure communication of DNS data by providing a means to check whether the data have been corrupted during communication over the Internet. While this detects the corruption of messages during communication, DNSSEC assumes that the zone private keys are not stolen and the servers that are responsible for providing authenticated name service are not corrupted themselves. Given the popularity of malicious attacks on DNS name servers, this is not a reasonable assumption to make.

The biggest dilemma in DNSSEC is where to store the zone private key, which is used to generate digital signatures. To prevent hackers from getting access to the zone private key, DNSSEC recommends to keep the key in a non-network connected, off-line, physically secure machine. But this creates a problem with providing support for dynamic updates in the domain name system because the off-line zone private key cannot be used to generate signatures (SIG RR) in real-time to authenticate dynamically updated data. This means dynamically updated data in DNSSEC are either not available or not protected before the next zone signing (using the zone private key) takes place.

There are other security problems with the DNS security extension. For example, expensive public key cryptography is used in DNSSEC to provide data integrity and source authentication. Therefore, to keep the cost down, signatures (SIG RR) in the DNS security extension are often generated with a long TTL (time-to-live). This opens a broad window for a freshness attack in which stale data whose corresponding signature has not expired is replayed to fool the client.

In addition, like non-secure DNS, DNSSEC cannot tolerate malicious or benign server failures. For example, newly updated data may be lost from the system if the primary name server of a zone fails before propagating the new information to the other name servers.

1.2 Our Contributions

To address the security issues of DNS, this thesis proposes a scalable Byzantine-fault-tolerant domain name system (SBFTDNS). A Byzantine-fault-tolerant system makes no assumption about the behavior of the faulty nodes [6]. The system provides high integrity, robustness, and availability of service in the presence of arbitrary failures, including failures due to malicious attacks. The proposed system consists of 3f+1 tightly coupled replicas per name server and guarantees safety and liveness properties of the system assuming no more than f replicas are faulty within a small window of vulnerability [9].

To authenticate communication between a client and a server, we propose the use of symmetric key cryptography. A session key, which is much shorter and less costly than a public key, is used to provide per-query data authentication and fresh data. The cost of establishing session keys using expensive public key operations is reduced by incorporating an efficient session key caching mechanism.

Symmetric cryptography requires that every communication pair share a session key, which could lead to servers needing to remember a huge number of keys unless care is taken care of. To address the scalability issues, we propose a hierarchical organization of name servers with a hybrid of the iterative and recursive query resolution approaches. This minimizes the number of keys to remember as well as reducing the load on the root name servers.

For performance and scalability, the system supports caching of session keys and resource records. However, caching in a security critical system raises concerns about cache consistency and staleness of data. We address this issue by designing a hierarchical cache with an invalidation mechanism using leases. Because of the use of hierarchical state partitioning and caching to achieve scalability in DNS, we can define an efficient protocol that allows replicas in a group to request operations from another group using very few messages.

1.3 What the System Offers

The newly proposed system significantly improves the security of the domain name system. The benefits come from two major sources.

- By using a Byzantine-fault-tolerant algorithm, the system tolerates server failures due to both benign faults and malicious attacks on servers. Unlike the DNS security extension, the system does not need to assume that the server private keys are not stolen, or the servers are not faulty.

- By using cryptographic operations, the system guarantees a secure communication mechanism by providing a way to detect whether DNS data has been corrupted during communication over the Internet.

By incorporating the Byzantine-fault-tolerant algorithm in our solution, we not only make the system hacker-tolerant, but also ensure correct name service even when a fraction of the replicas fail. Unlike the current insecure DNS and its security extension, our system tolerates server failures due to benign faults and malicious attacks. The system is reliable, i.e., the state of the system is not lost or corrupted when some node fails, and it is available, i.e., it continues to function normally even in the presence of failure. The system ensures consistency among data received from non-faulty name servers.

Moreover, with this scheme, real-time dynamic update is possible. Unlike the DNS security extension, the proposed system can store the server private keys online, since a hacker needs to compromise at least f+1 servers before compromising the system. This significantly improves the security of the dynamic update mechanism and makes dynamically updated data available in real-time. Furthermore, this offloads the administrators by getting them out of the loop.

By incorporating the session key mechanism, on the other hand, we provide per-query data authentication. Therefore, information provided by the servers is up-to-date and there is no stale data. Since we do not use pre-generated signatures, there is no possibility of freshness attacks until the signature expiration time, which is a problem in the DNS security extension.

We implemented the proposed system and compared its performance with non-secure DNS, and with TIS/DNSSEC, an implementation of the domain name security extension protocol. While our system performs worse than the insecure DNS due to extra cryptographic operations to support security, it performs as well or better than TIS/DNSSEC in almost every case, not to mention that we guarantee much higher level of security, robustness, and availability of the domain name system than TIS/DNSSEC. We observed a 4.2% gain in response latency for a typical A record query, and a 136% gain for an NS record query while using the proposed system instead of TIS/DNSSEC. The performance gain increases with the number of cryptographic operations, due to the elimination of expensive public-key cryptography and use of session-key based message authentication codes (MAC) in our system.

The newly proposed BFTDNS provides highly secure service, yet it is practical. It can not only provide secure name service for the Internet, but can also become a robust infrastructure for storing authenticated public keys. Other internetworking protocols that require authentication can leverage the open and widely available DNS security infrastructure, which adds to the attraction of a secure and scalable domain name system.

1.4 Thesis Outline

The rest of the thesis is organized as follows. Chapter 2 presents an overview of the domain name system infrastructure, i.e., the organization and the management of DNS. Chapter 3 discusses the security problems of current DNS and its security extension (DNSSEC). It also provides the motivation behind why a domain name system more secure than DNSSEC is needed. Chapter 4 describes the Byzantine-fault-tolerant algorithm we used in our system to implement a Byzantine-fault-tolerant domain name system. Chapter 5 describes the proposed design of a scalable Byzantine-fault-tolerant DNS (SBFTDNS) with a session key mechanism. Chapter 6 describes how we implemented SBFTDNS based on an existing DNSSEC package and a Byzantine-fault-tolerant replication library. Chapter 7 presents the experimental setup and the performance evaluation of SBFTDNS. Chapter 8 concludes the thesis and suggests future research directions.

Chapter 2

DNS Infrastructure

The abstract model of the domain name system [3] is a distributed database that maps between human readable hostnames, e.g., veena.lcs.mit.edu and machine-readable IP addresses, e.g., 18.26.0.72. DNS also provides other important information about the domain or host, e.g., email routing information, canonical name, etc. Different segments of the distributed database are locally administered, yet data in each segment is available across the network through a client-server scheme. Availability and performance of DNS are enhanced through a replication and caching mechanism. This chapter describes the basic layout and the organization of the domain name system.

2.1 Name Space

DNS name space is organized hierarchically. Like the UNIX filesystem, the whole DNS database can be viewed as an inverted tree, with the root node at the top. The root of the name space is a special node with a null label. All the other nodes are associated with a label of up to 63 characters. The domain name of a node in the tree is the list of labels starting from that node up to the root, using a period to separate each label, e.g., sarod.lcs.mit.edu. Every node in DNS tree must have a unique domain name since domain names are used as indexes into DNS database. The depth of the tree is limited to 127 levels.

A domain is simply a subtree of the domain name space with a name the same as the domain name of the node at the very top of the subtree. The top-level domains are divided into three areas:

1. generic top level domains (gTLDs), e.g .com, .edu, .gov, .mil, .net, .org, and .int. To accommodate the rapid expansion of the Internet, a few other gTLDs have been proposed, e.g., .web,. firm, .shop, .nom etc.
2. 2 character domains based on country codes found in ISO 3166, e.g., .uk is the top level domain for United Kingdom.
3. arpa is a special domain for the address to name mappings. For example, translating the address arpa.in-addr.40.0.26.18 (IP address 18.26.0.40) to the hostname chord.lcs.mit.edu

Figure 2-1 shows the hierarchical organization of DNS name space.

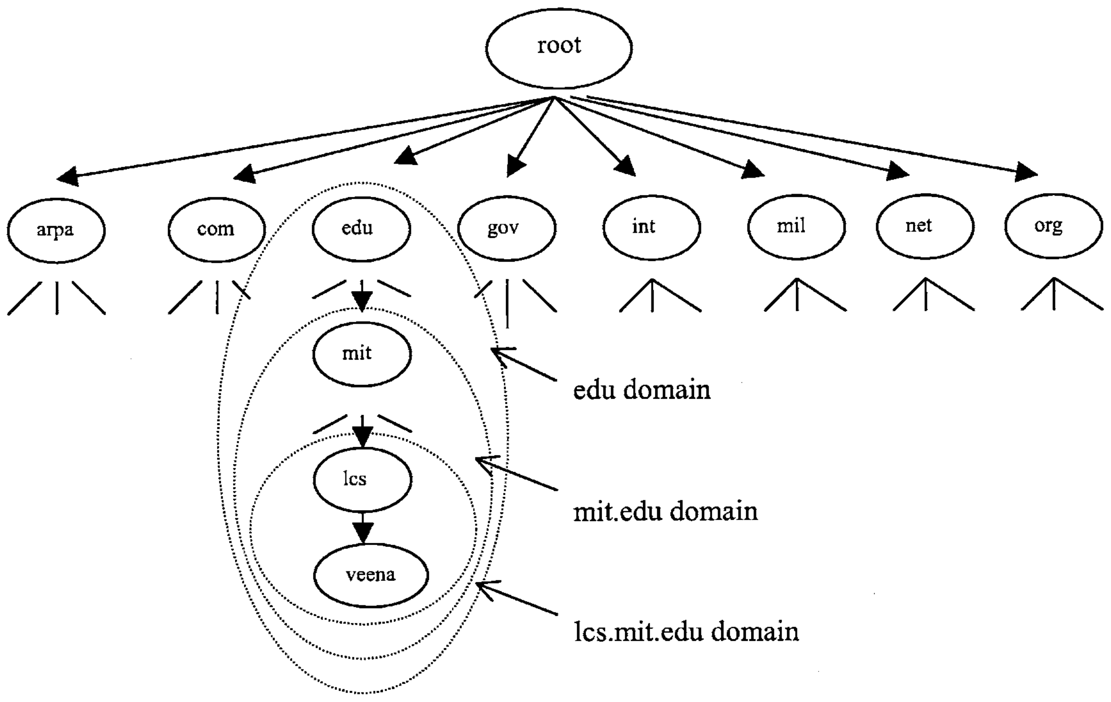


Figure 2-1: DNS Name Space Hierarchy

2.2 Administration

The domain name system is a distributed database. No single host on the Internet knows all DNS information. Similarly, administration of DNS is also hierarchical. This decentralized administration is achieved through delegation. No single entity manages every label in the tree. Instead, one entity maintains a portion of the tree, called a *zone* and delegates the responsibility to other entities for specific subzones.

A zone is a subtree of DNS tree that is administered separately. An organization gets the control of a zone by persuading the parent organization to delegate a sub-zone consisting of a single node. The parent organization does this by inserting a resource record in its database that marks the zone delegation. Every zone needs to provide a primary and a secondary name server, which contain the same information for the zone. A primary name server reads data for the zone from a file on its host. A secondary name server, on the other hand, gets zone data from an authoritative master server, which usually is the primary name server for the zone.

It is important to understand the difference between a zone and a domain. A zone contains domain names that the domain with the same domain name contains, except for domains named in delegated subdomains. For example, the toplevel domain *edu* contain the subdomains *mit.edu, cmu.edu, usc.edu* etc. Authority for each of these subdomains is delegated to the name servers of MIT, CMU and USC respectively. As shown in figure 2-2, while the **domain** *edu* contains all the data in *edu* plus all the data in *mit.edu, cmu.edu,* and *usc.edu,* the **zone** *edu* contains only the data in *edu,* which mostly comprises of pointers to the delegated subdomains.

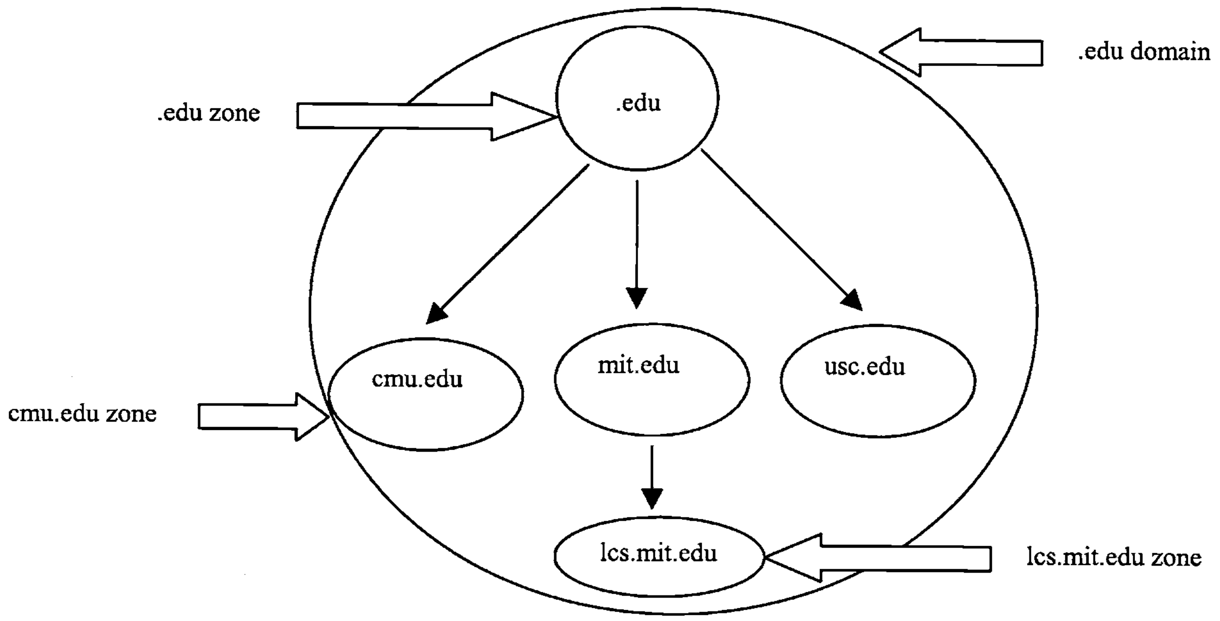


Figure 2-2: Distinction between a domain and a zone

2.3 Name Servers and Resolvers

The two major active components of DNS are name servers and resolvers. Name servers constitute the server half of DNS. Resolvers, on the other hand, are interfaces to end user programs.

2.3.1 Name Servers

The name servers are repositories of information and they answer queries using information stored in the database. They generally have complete information about a zone. The name server is then said to have authority for that zone. One name server can be authoritative for multiple zones too. Name servers listen to UDP and TCP port 53 for DNS queries. Every zone needs to provide a primary and a secondary name server, which contain the same information for the zone. Therefore, if one server is down, resolvers can use the other server. DNS incorporates this simple replication scheme to enhance:

- Redundancy
- Load Balancing
- Faster access to remote locations

Name Server Types

Primary servers get all the data for the zone from a locally stored file. All changes to a zone data must take place at the database of the primary server. Usually there is one primary server per zone.

Secondary servers get mirror copy of the zone data from another name server that is authoritative for the zone known as its master server. A secondary server periodically queries the master server and if there is any change in the database, the secondary server obtains the data and updates its own database. This transaction is known as the "zone transfer". There can be multiple secondary servers per zone.

Master Servers provide zone information to a requesting secondary server. This server can be a primary or another secondary server.

Caching only servers is not authoritative for any zone, as it does not have any local zone file; rather it relies on other name servers for authoritative answers. A caching only server merely accepts and processes requests, and stores the results in a cache for future resolutions.

Forwarders are name servers that are configured to accept requests for name resolution from other name servers, which fail to resolve a query locally. This is typically used within an intranet for name resolution that needs to go outside of the zone.

Root Name Servers

The root name servers know the names and addresses of the name servers for each of the top-level domains. In the absence of any other information, DNS query resolution has to start at the root name servers, which makes the root name servers the hotspot of DNS operation. To offload the root name servers, DNS uses caching extensively. However, the fact that any locally unresolved query is directed to the root name server keeps the roots extremely busy. Even with thirteen globally dispersed root name servers, the traffic to each root server is very high.

2.3.2 Resolvers

The resolvers are programs that interface user programs to the name servers. Resolvers perform the following functions

- Embody the algorithms required to resolve a query, i.e., find a name server that has the relevant information,
- Query the name server according to the end user request
- Interpret the responses from the name servers which may be the answer or an error message
- Return the answer back to the end user program

In most implementations, the resolver is a just a set of library routines with no cache of information. It places most of the burden of finding an answer on the name servers. This type of resolver is known as the *stub resolver* in DNS spec.

2.4 DNS Query Resolution

Resolution occurs when a client queries a name server for some host information, e.g., to get its IP address. If the name server in the local domain cannot resolve the client's request, it queries other servers to locate the server that can.

There are two types of DNS query resolution: recursive and iterative.

Recursive Query

In recursive resolution scheme, the burden of resolving a query is placed on a single name server. Once a name server receives a recursive query from a client, it is obliged to provide the client with the answer to the query, or an error if the answer could not be found. The name server, in recursive case, cannot refer the client to a different name server.

Iterative Query

In iterative resolution, a name server replies with the best answer it can give to the client, which may be another name server or a resolver with support for iterative query. The queried name server consults its own database for the queried data. If it cannot find the answer, it typically gives the IP address of the closest name server that might know the result of the query. The closest name servers are those authoritative name servers for a zone closest to the domain being looked up. Then the client repeats the request, this time sending it to the server it just learned about. By default, queries to root name servers are iterative.

A Typical DNS Resolution

Figure 2-3 desribes a typical DNS resolution for the IP Address of www.foo.edu.ca. In this case, a stub resolver sends a recursive query to a name server on the local host for the IP address of www.foo.ac.ca. The local name server searches its own database and cache. If it finds no entry for www.foo.ac.ca, it searches for the closest domain name foo.ac.ca, then ac.ca, and then for the entry for ca. Suppose the local name server does not find an entry for any of the previous domain names in its local cache.

At this point the local name server knows that the query cannot be resolved locally and it has to find the answer from other servers. Since it is not possible for one name server to know how to contact every other name servers, every name server is configured with a root cache file, which contains root domain name server names and addresses. There are thirteen such root name servers distributed all over the world. For example, the primary root name server A.ROOT-SERVERS.NET with IP address 198.41.0.4 is maintained by the Network Solutions Inc. and is located at Herndon, VA, USA.

As shown in figure 2-3, the local name server then iteratively queries a root name server for the IP address of www.foo.ac.ca and is referred to the ca name servers. The local name server then asks a ca name server for the desired address, which may refer it to the ac.ca name servers. The ac.ca server refers the local name server to the foo.ac.ca servers, which gives the IP address of www.foo.ac.ca. The local name server then returns the result to the stub resolver that issued the query.

If however, the local name server finds in its local cache an entry for any of the intermediate name servers, e.g., the IP address of the name server for the ac.ca zone, it can bypass contacting the root name server and can directly contact the ac.ca name server for the information it is looking for.

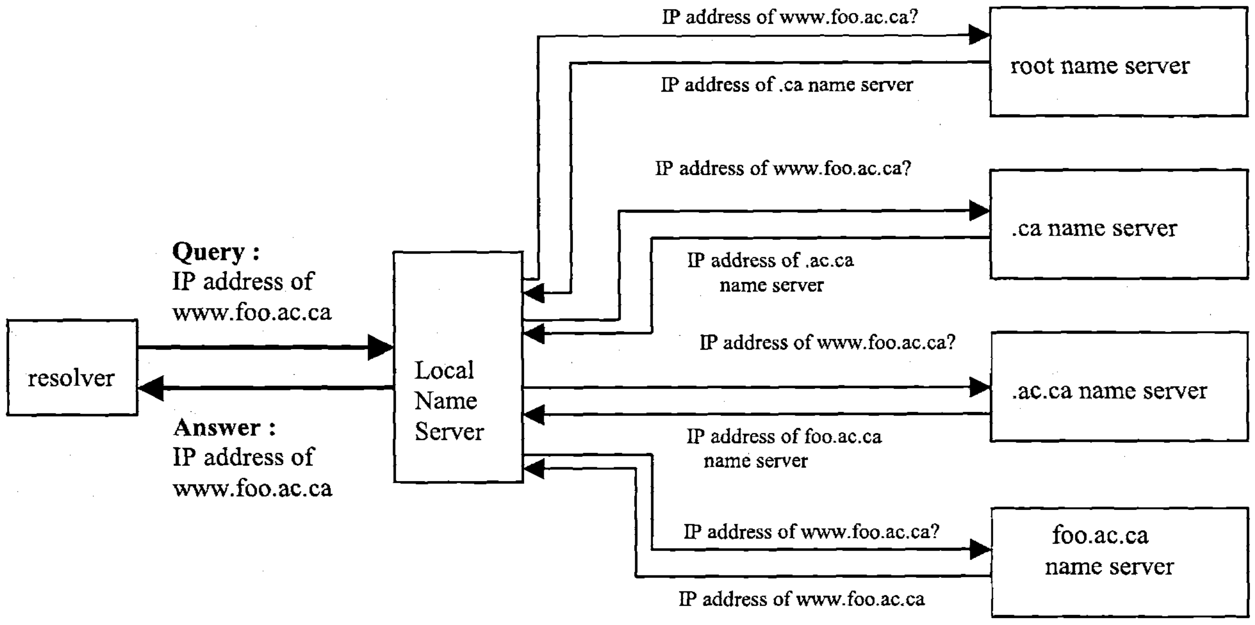


Figure 2-3: A Typical DNS Query Resolution for the IP Address of www.foo.edu.ca

2.5 Resource Records

The data associated with domain names are contained in resource records (RR). Each resource record carries a time-to-live (TTL), a type, and a class field followed by the data. All resource records with the same name, type and class but different values comprise a resource record set (RR Set).

RR Format

Each resource record has the following format

Name	TTL	Class	Type	RDLength	RData

The first field of an RR is the domain name. This is often called the *owner* of the RR.

The second field is a 32 bit unsigned integer that specifies the TTL (Time To Live), which is the length of the time the information in the RR is considered to be valid. This is the amount of time any name server can cache the data.

The third field indicates the RR *class,* which pertains to a type of network or software. RRs in today's DNS database mostly belong to the IN (Internet) class, although there are other classes like Chaosnet, or Hesiod.

The fourth field corresponds to the type of the RR as described in RFC 1035 [26]. There are about 40 different types of RR. The most commonly used types are as follow:

Type	Description
SOA	Start of authority
A	(IP) Address record
NS	Name Server record
CNAME	Canonical name (alias)
PTR	Pointer record
MX	Mail exchanger record

The Rdlength field specifies the length of the Rdata field.

Finally, the Rdata is a variable length string that describes the resource according to the format specified by the type and the class.

Standard RR Types

SOA Record

Each zone has exactly one SOA (Start Of Authority) record, which contains important information about the zone. Following is the SOA record for the mit.edu zone. We ignore the Rdlength field.

```
mit.edu.     6H IN SOA   BITSY.mit.edu. NETWORK-REQUEST.mit.edu. (
              2020            ; serial
              1H              ; refresh
              15M             ; retry
              5w6d16h         ; expire
              6H )            ; minimum
```

This SOA record indicates that for the mit.edu zone

- BITSY.mit.edu is the primary name server
- NETWORK-REQUEST@mit.edu is the contact email address for this zone
- 2020 is the serial number indicating the version number of DNS database
- 1 hour is the refresh period that indicates how often the secondary name server polls the master name server for updated information
- 15 minutes is the retry value that should elapse before a failed refresh should be retried
- 5 weeks 6 days 16 hours is the upper limit on the time interval that can elapse before the zone is no longer authoritative
- 6 Hours is the minimum TTL value for this zone.

NS Records

An NS (Name Server) record specifies a host that is authoritative for the specified domain and class. There are at least two NS records associated with a domain name. These are the delegation points of the parent-zone and the child-zones. For example:

```
mit.edu.    6H     IN     NS   STRAWB.mit.edu.
mit.edu.    6H     IN     NS   W20NS.mit.edu.
```

Usually, when an NS record is returned, the corresponding A record containing the IP address of the name server is also returned to avoid a second query for the IP address.

A Records

The most common RR is the address (A) record, which contains the name-to-address mappings. An A record associates an IP address to the domain name found in the first field of the record. For example:

```
veena.lcs.mit.edu        30M    IN     A      18.26.0.72
```

It is possible for a domain name to be associated with multiple IP addresses. This feature can be used to distribute and balance load between multiple servers. Conversely, different domain names can be associated with a single IP address. This makes it possible to use a single machine to act as virtual hosts to multiple services.

PTR Records

The PTR (Pointer) records encompass the address-to-name mappings, i.e., the reverse of the A records. They map an IP address to a human readable domain name. The IP addresses in PTR records are represented as a domain name in the in-addr.arpa domain. For example

```
72.0.26.18.in-addr.arpa 30M    IN     PTR    veena.lcs.mit.edu
```

MX Record

MX (Mail Exchange) records specify hosts that will accept mail for the specified domain. For example:

```
mit.edu.    6H    IN    MX   100    FORT-POINT-STATION.mit.edu.
mit.edu.    6H    IN    MX   100    PACIFIC-CARRIER-ANNEX.mit.edu.
mit.edu.    6H    IN    MX   100    SOUTH-STATION-ANNEX.mit.edu.
```

The number before the address is the preference of the mail exchange system. The lowest preference value is tried first and equally weighted entries are tried randomly.

CName Record

A CNAME (Canonical Name) RR defines an alias for the host showed in the data field. For example:

```
ftp.musenet.org      30M      IN    CNAME   cyberion.musenet.org
```

This example defines "ftp.musenet.org" as an alias for "cyberion.musenet.org".

Chapter 3

DNS Security Issue

3.1 Problems with current DNS

The original designers of DNS did not realize that DNS would evolve so fast to become the basis of the Internet. They did not design an adequately secure DNS protocol. As a consequence, the domain name system, as it stands today, suffers from some intrinsic security problems. The major security holes in current DNS infrastructure include lack of authentication, lack of a consistency control mechanism, and vulnerability to server failures due to both benign faults (e.g., a server crash) and malicious attacks on servers.

3.1.1 Lack Of Authentication

The biggest security hole in the current domain name system is the lack of data integrity authentication, source authentication, and authorization. DNS servers never authenticate the data they send to the clients. Clients can only judge the origin of the reply data from the IP address of the sender included in the reply data, which is very easy to fake. Therefore, nothing prevents a hacked name server from sending incorrect data, spoofing network data, stealing or redirecting valuable data, and nothing prevents the resolvers from trusting false information. In addition, there is no guarantee that the servers and the clients performing the transactions are the entities they claim to be.

For example, a possible way to attack DNS by taking advantage of the lack of data integrity and source authentication would be to take control over a router and then spoof network data packets containing the answer to a DNS query that pass through the router. Figure 3-1 illustrates this scenario.

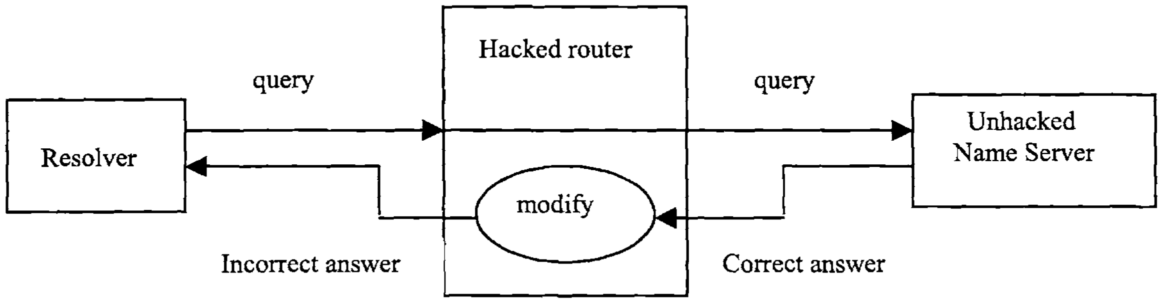


Figure 3-1: DNS attack by breaking into a router

Another way a malicious user can attack without controlling a router is to sit in the way a query packet passes, sniff the packet, and then generate a wrong answer fast enough so that the fake answer reaches the resolver before the correct answer from the originally intended name server. The source IP address and source port fields should be set properly so that the resolver thinks the packet is from the name server it originally sent the request to. This is known as the 'man-in-the-middle- attack. Figure 3-2 illustrates such an attack.

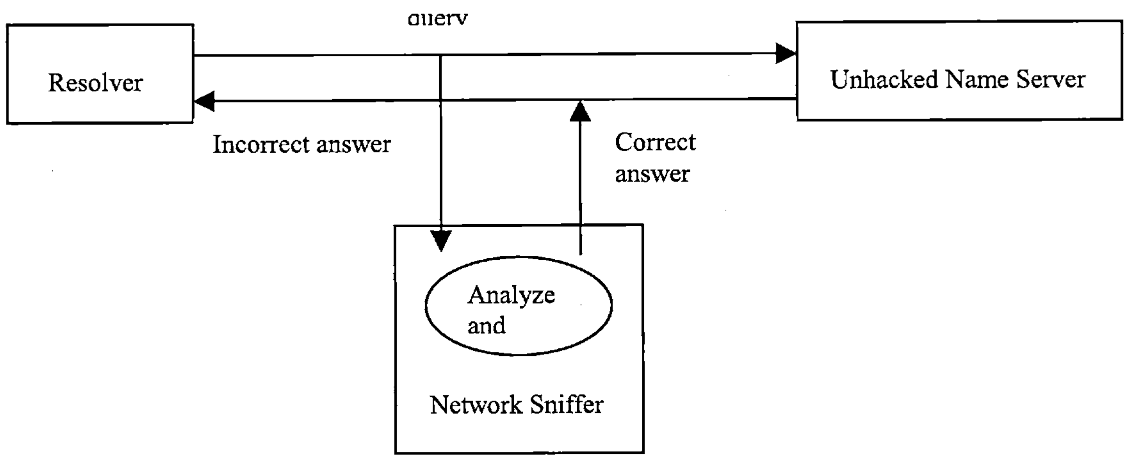


Figure 3-2: Man-in-the-middle attack of DNS by network packet spoofing

Also, without any authentication and authorization scheme, it is possible to issue a fraudulent update request to the zone administrator to change some resource records, and thus corrupt DNS database, as illustrated in figure 3-3. Because of the decentralized administration of DNS, achieved through zone delegation, the root user of the host veena.lcs.mit.edu may change the IP address of the host and notify the administrator of the lcs.mit.edu zone of the update. As it stands now, there is no way to authenticate the update request from the message itself in current DNS. For a local case, the administration can ensure the authenticity of the update request by some manual means, e.g., a phone call. This, however, defeats the purpose of the decentralized administration of the domain name system and proves impractical for a huge entity like the Internet.

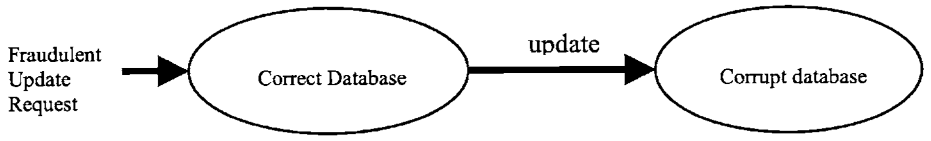


Figure 3-3: DNS attack by issuing fraudulent update request

3.1.2 Lack of Consistency Control

DNS uses a simple primary-secondary scheme to ensure service availability and load distribution among the servers. The secondary name server splits the load with the primary server or handles the whole load if the primary is down; clients contact only one server at a time. While the primary server gets its data from a file, the secondary server loads its data over the network from a master name server during a *zone transfer,* the period of which is determined by the zone administrator. During updates, the data only gets updated to the primary files. Therefore, the states of the primary name server differ from that of the secondary after an update and before the next zone transfer. Especially in highly dynamic zones, the secondary name server almost always contains some stale data and there is inconsistency in the results served by the primary and the secondary name server. This causes consistency and reliability problems even without an intentional attack since clients may get stale data from some servers. Moreover, some newly updated data may be lost from the system forever if the primary name server crashes before the data is transferred to the other servers. DNS therefore does not tolerate benign faults at the servers.

3.1.3 Vulnerability to Server Failures

In current DNS infrastructure, only one name server gets contacted at a time to provide the name service. Therefore, one compromised server is enough to block information, steal or redirect valuable information, or fool the resolvers with incorrect information. Figure 3-4 describes a scenario where an intruder breaks into a name server and fools any subsequent client that contacts this compromised server with incorrect answers to DNS queries.

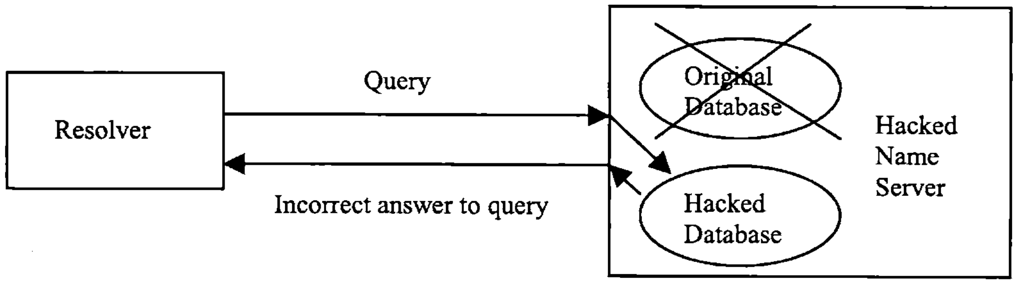


Figure 3-4: DNS attack by a Single Name Server Break In

In the worst case, an intruder breaks into the primary name server of a zone and corrupts the database. The secondary name server contacts the primary periodically (according to the refresh value in the zone SOA RR) and downloads any new data. As there is no way for the secondary name server to authenticate the validity of the data provided by the primary name server, it trusts every byte it gets from the primary and therefore, both the databases get corrupted.

3.2 Proposed DNS Security Extension (DNSSEC)

In late 1997, a security extension of DNS (DNSSEC) was proposed [14][17] by the Internet Engineering Task Force (IETF) to make DNS infrastructure more secure and robust. This extension was partially implemented by Trusted Information Systems, Inc. and an experimental version of it has recently been incorporated into BIND version 9.0 [20]. The basic idea behind DNSSEC is to provide data authentication to security aware resolvers and applications through the use of cryptographic digital signatures. DNSSEC provides three distinct services [11]

- key distribution
- data integrity and origin authentication
- transaction and request authentication.

To provide authentication, DNSSEC introduced some new RR types in DNS.

3.2.1 New RR Types

KEY Resource Record

The KEY RR is used to store the public key that is associated with a domain name. This can be the public key of the zone, host, user or some other entity. Usually, there is a single private key per zone, but there might be multiple keys for different algorithms, signers, etc. To construct a trusted authentication chain, a secure zone must contain a KEY RR for every delegated subzone that is signed by the private key of the superzone.

A resolver could learn the public key of a zone either by having it be statically configured within it, or by making a DNS query. To reliably learn a public key by querying DNS, the key itself must be signed with a key the resolver can authenticate.

Once a security aware resolver reliably learns the public key of the zone, it can authenticate whether signed data read from that zone is properly authorized or not. For utmost security, DNSSEC recommends storing the zone private key offline and using it to re-sign all the records in the zone periodically. However, there are cases, for example dynamic updates [13], where the zone private key needs to be kept online.

SIG Resource Records

The signature resource record (SIG RR) is the primary tool to ensure data integrity and authentication in DNSSEC. A SIG RR unforgeably authenticates an RRset (a set of resource records with the same name, type and class) by binding it to the signer's domain name and a validity interval. This is done using cryptographic techniques and the signer's private key. In most cases, the signer is the owner of the zone from which the RR originated.

Every name in a secure zone is associated with at least one SIG RR for each resource record type under that name except for glue address records and delegation point NS RRs [14]. A security aware name server returns, with the RRs retrieved, the corresponding SIG RR, which is used by a security aware client to authenticate the returned RRset.

NXT Resource Records

The next (NXT) RR is used to securely indicate that RRs with an owner name in a certain name interval do not exist in a zone and to indicate the RR types present under an existing name. This is important because in DNSSEC, a SIG RR can only be used to sign existing RRsets in a zone. However, this is insufficient to prevent the authenticated denial of DNS entries. Only with the NXT RR, it is possible to provide "data origin" authentication for the non-existence of a domain name in a zone or the non-existence of a type for an existing name. For example, the NXT RR with the value

```
fix.foo.bar    500    IN    NXT    frodo.foo.bar    NS    SIG    KEY    NXT
```

indicates that there is no domain name that existing lexicographically between fix.foo.bar and frodo.foo.bar. Also, fix.foo.bar has NS RR, SIG RR, KEY RR, and NXT RR under its name.

3.2.2 The Chain of Authentication

When a security aware resolver gets a response from a security aware name server, it verifies the signature for each RRset in the response. From the resolver's point of view, a verified signature establishes the integrity of data. However, the resolver must know whether it should trust the KEY used to sign the data and whether that KEY was permitted to sign that message. A security aware resolver, therefore, needs to establish a cryptographically verified path from a known trusted point to the point represented by the name in the response. Since the SIG RR associated with the KEY set is signed by the parent zone's key, the resolver will request security information of the parent. This process will continue further up the tree until the resolver finds a trusted key. In the worst case, this will end with the root server key, which is statically configured in all DNSSEC aware entities [14][17]. In this way, a security aware resolver establishes the authenticity of the source by "walking DNS chain of trust".

Figure 3-5 describes a sample resolution for the IP address of veena.lcs.mit.edu in DNSSEC. The notation $(X)_K$ indicates that contents of the message X is digitally signed by the key K. For this example, we assume that the resolver is capable of making a recursive query. The key of the local name server K_{local} is statically configured at the resolver. When the resolver asks the local name server for the IP address of veena.lcs.mit.edu, the local name server, being unable to resolve the query using its local database and cache, gives the IP address and the key of the root name server (K_{root}) to the resolver. To prove authenticity of the answer, the local name server signs the data it sends to the resolver using the key K_{local}. Upon verification, the resolver directs the query for the IP address of veena.lcs.mit.edu to the root name server. The root name server gives the resolver the IP address of the name server of the mit.edu zone and its key K_{mit}. When the resolver verifies that the result it got from the root name server is signed by the trusted key K_{root}, it trusts the authenticity of the response, and sends the query for veena.lcs.mit.edu to the name server of the mit.edu zone. The name server of the mit.edu zone provides the resolver with the IP address of the lcs.mit.edu zone and its key $K_{lcs.mit}$. Upon successful verification of the signed response using the key K_{mit}, the resolver directs the query for the

IP address of veena.lcs.mit.edu to the name server of lcs.mit.edu zone. The name server of the lcs.mit.edu zone finally provides the resolver with the IP address of veena.lcs.mit.edu. To prove authenticity, the answer is signed using the key $K_{lcs.mit}$. Upon successful verification, the resolver trusts the final answer it receives from the name server of the lcs.mit.edu zone. Figure 3-5 illustrates the resolution scheme described above.

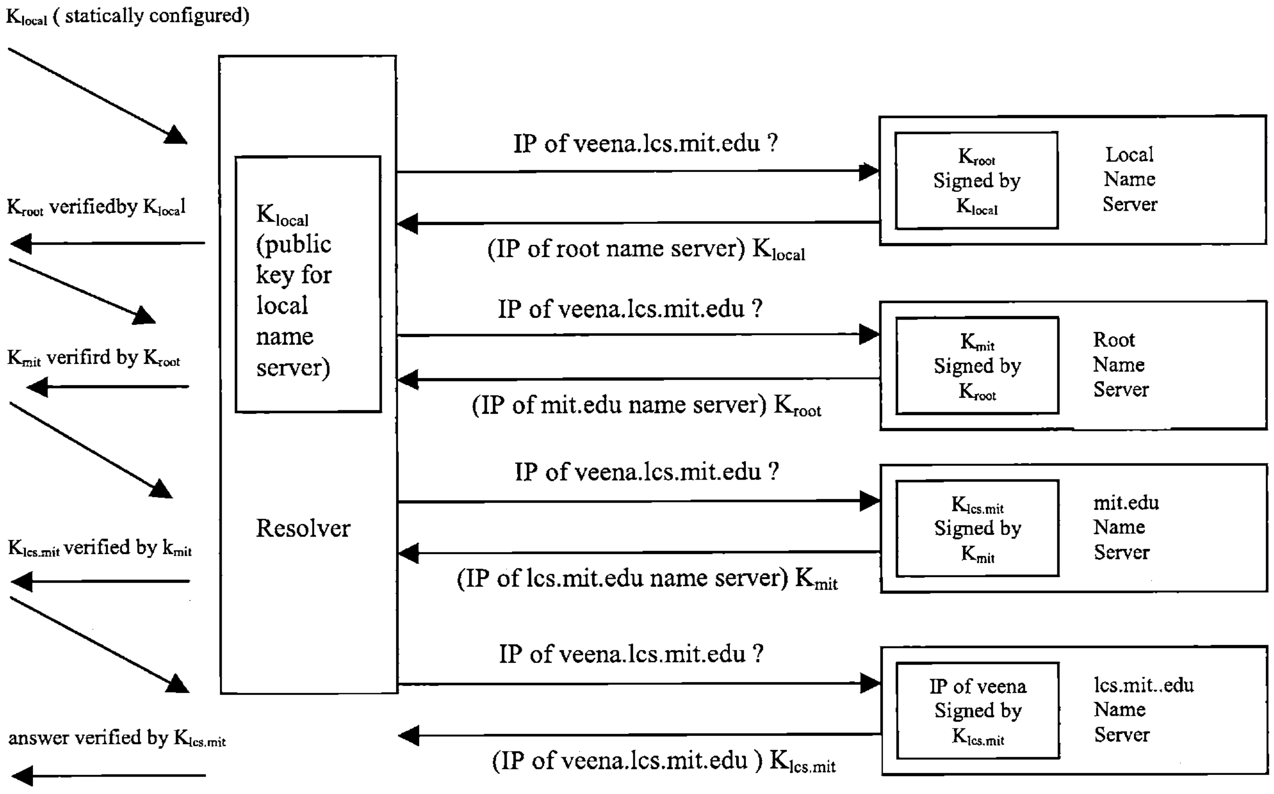

K_local (statically configured)
K_root verifiedby K_local
K_mit verifird by K_root
K_lcs.mit verified by k_mit
answer verified by K_lcs.mit
K_local (public key for local name server)
Resolver
IP of veena.lcs.mit.edu ?
(IP of root name server) K_local
IP of veena.lcs.mit.edu ?
(IP of mit.edu name server) K_root
IP of veena.lcs.mit.edu ?
(IP of lcs.mit.edu name server) K_mit
IP of veena.lcs.mit.edu ?
(IP of veena.lcs.mit.edu) K_lcs.mit
K_root Signed by K_local
Local Name Server
K_mit Signed by K_root
Root Name Server
K_lcs.mit Signed by K_mit
mit.edu Name Server
IP of veena Signed by K_lcs.mit
lcs.mit..edu Name Server

Figure 3-5: A Sample DNS Query Resolution in DNSSEC

3.2.3 Dynamic Update

To support dynamic update of data, the secure DNS extension has defined a new DNS opcode, new DNS request and response structure, and new error codes [13]. All the data in a secure zone is signed either by a zone key or by a dynamic update key tracing its authority to a zone key. An update is defined to be any combination of deletion and insertion of resource records with one or more owner names; however, all the changes for any particular DNS update are restricted to a single zone. Updates occur only at the primary server of a zone. A dynamic secure zone is any secure zone that can interpret an update request, verify the authentication and authorization of the request, and update the RR database in *realtime*.

RFC 2137 [13]describes two basic modes of dynamic secure updates, mode A and mode B. A summary of comparison table is given below.

Criteria	**Mode A**	**Mode B**
Definition	Zone key **offline**	Zone key **online**
Server Workload	Medium	High
Dynamic Data Temporality	Transient	Permanent

Mode A

In mode A, the zone owner private key and the static zone master file are kept offline for maximum security of the static zone contents. Their authorizing dynamic key signs any dynamically added or updated data and they are backed up, along with the SIG RR, in a separate online dynamic master file. This mode reduces server computation as the server only needs to check the signatures on the update request and data, which have already been signed by the updater (generally checking signature is a much faster operation than signing the data), and update the relevant NXT RR if needed. Since the dynamic data is only stored in an online dynamic master file and is only authenticated by the dynamic update keys that may expire, updates in mode A are transient in nature.

Mode B

In mode B, the zone owner private key and the master file are kept online at the zone primary server. After a successful, authenticated update, the SIG RRs under the zone key for the resulting data are generated and these SIG and possible SOA/NXT changes are entered into the zone and the unified online master file. This mode, therefore, requires more effort on the server's part as it needs to compute zone data signatures in addition to verifying the signatures on the request. Since signing generally takes more time than verification, the server needs to do more computational work than it would do to verify the data signatures required in mode A. For mode B, the incorporation of the updates into the primary master file and their authentication by the zone private key make them permanent in nature.

3.3 Problems with DNSSEC

While the DNS security extension provides data integrity and source authentication, it still suffers from some security flaws. DNSSEC ensures a secure communication of DNS data by providing a means to check whether data has been corrupted during communication over the Internet. While this detects the corruption of messages during communication, DNSSEC does not tolerate server failures due to benign faults or malicious attacks. DNSSEC assumes that the zone private keys are not stolen and the servers that are responsible to provide authenticated name service are not corrupted themselves. Given the popularity of malicious attacks on the name servers, this is not a reasonable assumption to make.

The biggest dilemma in DNSSEC is where to store the zone private key. The other problems include DNSSEC's vulnerability to server failure, freshness attack, and the lack of a consistency control mechanism.

3.3.1 Zone Private Key Storage

Online Storage

The biggest advantage of storing the zone private key online, as in mode B, is that new SIG RRs can be calculated immediately after an authenticated update succeeds. This ensures a *real-time* dynamic update. However, if the zone private key is stored online, then a security breach of the server that stores the zone private key gives an intruder full access to the zone, i.e., he can sign whatever he wants, issue fraudulent update requests, and corrupt the database. This mode therefore cannot survive the security breach of a single server.

Storing an encrypted private key in the server and decrypting it before use also doesn't work, because the intruder may dump and analyze the memory to retrieve the private key once he gets hold of the server that stores the zone private key. For a system like this, the zone private key should never be in the server memory even for the shortest period to ensure full security.

One might consider the use of special hardware like a smartcard that is easy to attach or detach from a computer to store the zone private key. This needs special hardware support. Moreover, the moment someone attaches the smartcard to a compromised server, it is possible to use the password stolen from the system to get the smartcard to sign fraudulent update requests or sign bad RRs, even though the intruder may not know the zone private key. Furthermore, it is not absolutely impossible to steal the private key stored in a smartcard.

The zone private key can also be stored in a secure coprocessor attached to the server. This ensures that the zone key cannot be stolen from the secure coprocessor. However, a compromised server can get any data signed by the zone private key without a need to expose or steal the zone key from the coprocessor.

Offline Storage

When dynamic update keys are stored online, as in mode A, the dynamic data stored only in an online dynamic master file renders updates to be transient in nature, which is not desirable. Moreover, the solution of keeping the zone private key offline while keeping the dynamic update keys online is no different from keeping the zone key online, because by breaking into a server that holds the dynamic update keys, an intruder may get hold of the keys and issue fraudulent update requests that will corrupt the database.

For utmost security, DNSSEC recommends that zone private key should be stored in a offline, non-network connected, physically secure machine, and a portable media, such as a diskette, should be used to transfer the update request and the signed new data between the signer and the primary name server. This, however, defeats the purpose of real-time, dynamic update because it is not possible to get a dynamically added or changed resource record signed right away if the zone private key is stored offline. Such data is either not available or not protected by the zone private key before the next signing takes place, which occurs infrequently. This may not be harmless as update requests are often time critical. For example, if some user notices that her private key is compromised, she would immediately want to revoke the corresponding KEY and SIG RRs. If, however, the zone private key is kept offline, she may have to wait until the next signing point. In the mean time, the intruder is free to abuse the compromised key.

3.3.2 Freshness Attack due to Pre-generated SIG RR

Another serious problem with DNSSEC is its vulnerability to freshness attacks. A freshness or replay attack occurs when a message or a message component for a previous run of a protocol is recorded by an intruder and replayed in a later run of the protocol.

DNSSEC provides an unforgeable authentication of an RRset of a particular type, class, and name by associating it with a signature resource record that binds DNS data to a time interval and the signer's domain name. However, since signature generation is expensive, SIG RRs are not generated on a per-query basis; rather they are computed occasionally, and are valid from a pre-set signature inception time to a signature expiration time, and this interval tends to be long. The original TTL (time-to-live) value of the RR is also included and protected by the signature. The corresponding RR is valid until the signature expiration time or the TTL expiration time, whichever comes first. This opens a big window for freshness attacks. Even if some RR is updated, the actual update remains ineffective until the signature expiration time. In the mean time, a malicious user can replay the stale data to fool a client.

3.3.3 Lack of Consistency Control and Vulnerability to a Single Server Failure

the DNS security extension attempts to patch the biggest security hole in DNS infrastructure by providing data integrity and origin authentication. However, it still suffers from the lack of consistency control and vulnerability to a single server failure due to both benign faults (e.g., a server crash) and malicious attacks.

DNSSEC uses a simple primary-secondary replica scheme to provide server availability and load distribution, as in the non-secure version of DNS. Only one of the primary or the secondary name server takes part in a particular query. Data update takes place only at the primary server and the secondary server knows nothing about the update until after the next zone transfer. Thus, there may be inconsistencies between the states of the primary and the secondary server, and a client query may result in completely different answers depending on which server served the query. This causes consistency and reliability problems even without an intentional attack. Moreover, if the primary server crashes and loses state before the secondary server loads the updated data from primary, the newly added or updated data is lost forever. DNSSEC, therefore, does not tolerate benign server faults.

Also, like the non-secure DNS, only one name server gets contacted at a time to provide the name service in DNSSEC. Therefore, one compromised server is enough to block information, steal valuable information, redirect information, issue fraudulent update requests, or fool the client with wrong information in DNSSEC.

3.4 Why We Need Something More Secure than DNSSEC

From the above discussion, it is apparent that the IETF proposed secure extension of the domain name system (DNSSEC) is not secure enough, especially in a highly dynamic environment. With its exponential growth in size and popularity, the Internet today is more dynamic than ever. Thousands of hosts are joining the Internet everyday. According to a domain survey done by the Internet Software Consortium available at http://www.isc.org/ds/WWW-200007/index.html [19], the number of hosts advertised on DNS was 56,218,000 in July 1999. By January 2000, the host count went up to 72,398,092. According to the latest survey in July 2000, the host count in DNS was 93,047,785. In addition, server load distribution schemes and mobile computing, which tend to use highly dynamic DNS data [35], are getting very popular. This trend, together with the prospects of DNS serving as a highly reliable public key infrastructure for a variety of internetworking protocols and applications, calls for a secure and robust domain name system.

The IETF proposed DNS security extension (DNSSEC) uses end-to-end cryptographic techniques to ensure a secure communication of DNS data. While end-to-end [32] cryptographic techniques detect the corruption of messages during communication, cryptography alone cannot make a system tolerant against server failures, both due to benign faults and malicious attacks. DNSSEC assumes that the zone private keys are not stolen and the name servers are not corrupted themselves. However, given the popularity of malicious attacks on the name servers, it is not reasonable to make such assumptions. Moreover, as discussed in section 3.3, DNSSEC cannot tolerate some benign server failures. To ensure high integrity, robustness, and availability of the system in the presence of arbitrary failures in an asynchronous environment like the Internet, we need Byzantine fault tolerance [22], i.e., resistance of the system to arbitrary misbehavior of faulty components [6]. Cryptography and Byzantine fault tolerance together can solve the security problems of the domain name system. This is the approach we take in designing a secure and robust DNS.

Chapter 4

A Practical Byzantine Fault Tolerant Algorithm

A Byzantine-fault-tolerant system makes no assumption about the behavior of the faulty nodes [6]. In a highly asynchronous system like the Internet in the presence of malicious attacks, it is important that a system is Byzantine fault tolerant, i.e., the system provides high integrity, robustness, and availability of service in the presence of arbitrary failures. In this chapter, we describe a Byzantine fault tolerant algorithm that we consider to be practical and suitable for a system like ours. The algorithm is developed by Miguel Castro and Barbara Liskov of the programming methodology group of MIT Laboratory of Computer Science [6][7][8][9]. The system consists of 3f+1 tightly coupled replicas per name server and guarantees safety and liveness properties of the system assuming no more than f replicas are faulty within a small window of vulnerability [9].

4.1 The CLBFT Algorithm

M. Castro and B. Liskov's practical Byzantine fault tolerant algorithm (CLBFT) provides asynchronous state-machine replication that offers both integrity and high availability in the presence of Byzantine faults. The approach is interesting for two reasons: (1) it improves security by recovering replicas proactively, and (2) it is based on symmetric key rather than expensive public key cryptography, which allows it to perform well so that it can be used in practice to implement real services.

Together with its recovery mechanism, the algorithm allows a system to tolerate any number of faults over the lifetime of the system, provided fewer than 1/3 of the replicas are faulty within a window of vulnerability that is small under normal conditions. The window may increase under a denial of service attack, but the algorithms can detect and respond to such attacks, thereby maintaining the integrity of the system.

4.2 The System Model and Assumptions of the CLBFT

The CLBFT assumes an asynchronous distributed system where nodes are connected by a network. The network may fail to deliver messages, duplicate them, or deliver them out of order.

The CLBFT uses a Byzantine failure model, i.e., faulty nodes may behave arbitrarily, subject only to the following restrictions:

- Node failures are independent, i.e., one node failure does not necessarily cause other node failures.
- The adversary and the nodes it controls have bounded computational power such that it is unable to subvert the cryptographic techniques used in the system.
- The adversary cannot delay correct nodes indefinitely.

Cryptographic techniques are used to prevent spoofing and replays and to detect corrupt messages. The CLBFT makes use of public-key cryptography [30], message authentication codes [29], and message digests produced by collision resistant hash functions.

4.3 How the CLBFT Algorithm Works

The CLBFT algorithm is a form of state machine replication. The service is implemented by a set of replicas R and each replica is identified using an integer in $\{0,\ldots,|R|-1\}$. Each replica maintains a copy of the service state and implements the service operation. The algorithm assumes $|R| = 3f + 1$ where f is the maximum number of replicas that may be faulty. Service nodes are non-faulty if they follow the algorithm and no attacker can impersonate them (e.g., by forging their MACs).

To tolerate Byzantine faults, every step taken by a node is based on obtaining a certificate, which is a set of messages certifying some statement is correct and coming from different replicas. The size of the set of messages in a certificate is either f+1 or 2f+1, depending on the type of statement and step being taken. A certificate of size f+1 is sufficient to prove that the statement is correct because it contains at least one message from a non-faulty replica. A certificate of size 2f+1 ensures that it will also be possible to convince other replicas of the validity of the statement even when f replicas are faulty.

To guarantee safety, all non-faulty replicas need to agree on a total order for the execution of requests despite failures. The system ensures this behavior by using a primary-backup mechanism where replicas move through a succession of configurations called views. In a view, one replica is designated as the primary and the others are backups. The algorithm chooses the primary p of a view v such that $p = v \bmod |R|$ where views are numbered consecutively. View changes are carried out to provide liveness by allowing the system to make progress when the current primary fails.

The algorithm works roughly as follows:

1. A client sends a request to invoke a service operation to the primary.
2. The primary multicasts the request to the backups.
3. The replicas execute the request and send a reply to the client.
4. The client waits for f+1 replies from different replicas with the same result; this is the result of the operation.

When the primary receives a request, it uses a three-phase protocol to atomically multicast the request to the backups. The three phases are *pre-prepare, prepare* and *commit.* The pre-prepare and prepare phases are used to totally order requests sent in the same view even when the primary, which proposes the ordering of requests, is faulty. The prepare and commit phases are used to ensure that requests that commit are totally ordered across views. All the messages between the replicas in this three-phase protocol are authenticated using MACs. Figure 4-1 shows the operation of the algorithm in the normal case of no primary faults.

Each replica stores the service state, a log containing information about requests, and an integer denoting the replica's current view. The log is kept for recovery purpose.

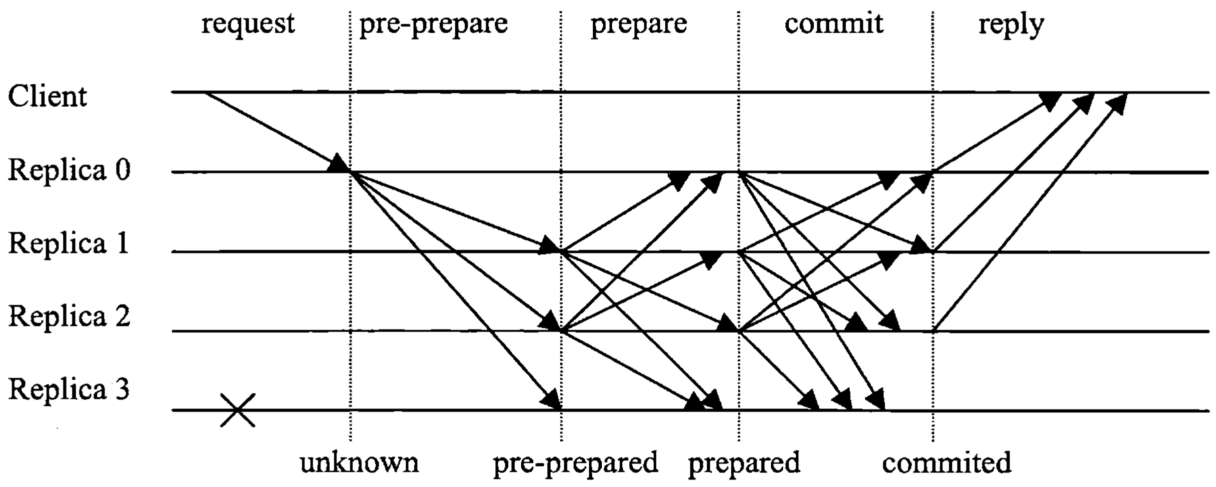


Figure 4-1: Normal case operation in CLBFT. Replica 0 is the primary, replica 3 is faulty

Replicas can discard entries from the log once the corresponding requests have been executed by at least f+1 non-faulty replicas, a condition required to ensure that request will be known after a view change. The algorithm reduces the cost by determining the condition only when a request with a sequence number divisible by some constant K (e.g., K = 128) is executed. The state produced by the execution of such requests is termed as *checkpoints.* When a replica produces a checkpoint, it multicasts the checkpoint message to other replicas and waits until it has a certificate with 2f+1 valid checkpoint messages for sequence number n of the last request whose execution is reflected in the state with the same state digest d sent by different replicas. At this point the checkpoint is known to be stable and the replica *garbage collects* all entries in its log with sequence numbers less than or equal to n; it also discards earlier checkpoints.

4.3.1 View Changes

The CLBFT algorithm uses a view change protocol to ensure *liveness* by allowing the system to make progress when the current primary fails [9]. The protocol also preserves *safety*: it ensures that non-faulty replicas agree on the sequence numbers of committed requests across views.

View changes are triggered by timeouts that prevent backups from waiting indefinitely for requests to execute. A backup is *waiting* for a request if it received a valid request and has not executed it. A backup starts a timer when it receives a request and the timer is not already running. It stops the timer when it is no longer waiting to execute the request, but restarts it if at that point it is waiting to execute some other request.

If the timer of backup i expires in view v, the backup starts a view change to move the system to view $v + 1$. It stops accepting messages (other than checkpoint, view-change, and new-view messages) and multicasts a *view-change* message to all replicas.

The new primary p for view $v + 1$ collects a quorum certificate with 2f+1 valid *view-change* messages for view $v + 1$ signed by different replicas. After obtaining the new-view certificate and making necessary updates to its log, p multicasts a *new-view* message to all other replicas, and enters view $v + 1$: at this point it is able to accept messages for view $v + 1$. A backup accepts a new-view message for $v + 1$ if it is properly signed, if it contains a valid new-view certificate, and if the message sequence number assignments do not conflict with requests that committed in previous views. The backup then enters the view $v + 1$, and becomes ready to accept messages for this new view.

Figure 4-2 illustrates an instance of the view-change protocol. The detail of the protocol can be found at [9].

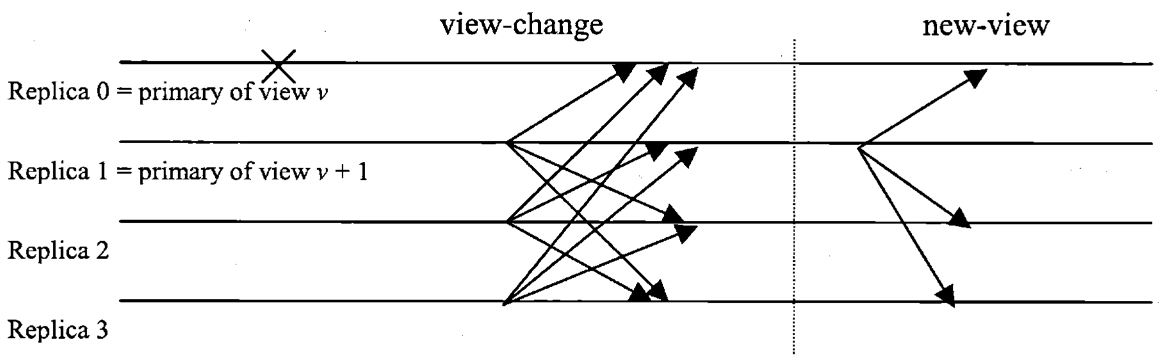

view-change
new-view
Replica 0 = primary of view v
Replica 1 = primary of view v + 1
Replica 2
Replica 3

Figure 4-2: View Change Protocol

4.3.2 Proactive Recovery

The recovery protocol makes faulty replicas behave correctly again to allow the system to tolerate more than f faults over its lifetime. To achieve this, the CLBFT protocol ensures that after a replica recovers, it is running correct code, it cannot be impersonated by an attacker, and it has correct, up-to-date state.

Since a Byzantine-faulty replica may appear to behave properly even when broken, the recovery mechanism is proactive to prevent an attacker from compromising the service by corrupting 1/3 of the replicas without being detected. The algorithm recovers replicas periodically independent of any failure detection mechanism. A recovery monitor saves the replica's state to disk, reboots the system with correct code and restarts the replica from the saved state. The correctness of the operating system and service code is ensured by storing them in a read-only medium. At this point, the replica's code is correct and it did not lose its state. The replica must retain its state and use it to process requests even while it is recovering. This is vital to ensure both safety and liveness in the common case when the recovering replica is non-faulty; otherwise recovery could cause the f+1st fault.

When a node have been attacked, the attacker is not able to steal the node's private key because this is stored in a secure coprocessor; the details are discussed in [9]. However, the attacker is able to steal all the secret keys known to that node, and furthermore, it can cause the coprocessor to sign lots of erroneous messages. Therefore, if the recovering replica *r* was faulty, the state may be corrupt and the attacker may forge messages because it knows the MAC keys used to authenticate both incoming and outgoing messages. To overcome these problems, the recovering replica *r* discards the keys it shares with clients and it multicasts a new-key message to change the keys it uses to authenticate messages sent by the other replicas. This is important if *r* was faulty because otherwise the attacker could prevent a successful recovery by impersonating any client or replica. The algorithm allows the replica to continue participating in the request processing protocol while it is recovering, since this is sometimes required for it to complete the recovery. The detail of the recovery protocol can be found in [9].

4.4 The Properties of the CLBFT Algorithm

The service provided by the CLBFT algorithm is modeled as a state machine that is replicated across different nodes in a distributed system. The algorithm can be used to implement any replicated service with a state and some operations. The clients issue requests to the replicas and block waiting for the reply. Like all state machine replication techniques, the CLBFT imposes two restrictions on the replicas: they must start in the same state, and they must be deterministic. The system can allow some common forms of non-determinism [6].

The CLBFT ensures *safety* for an execution assuming no more than f replicas out of a total of 3f+1 replicas are faulty within a window of vulnerability. *Safety* means that replicated service satisfies *linearizability* [6] i.e., service behaves like a centralized implementation that executes operations atomically one at a time. The system provides safety irrespective of the number of faulty clients using the system, even if they collude with faulty replicas. In DNS, a faulty client can issue an unauthorized update request. This is checked by the authentication mechanism provided by the CLBFT, which ensures that the replicas will authenticate the client and can deny an unauthorized request issued by faulty client.

The algorithm also guarantees *liveness* [6] i.e., non-faulty clients eventually receive the service, provided (1) at most f replicas are faulty within the windows of vulnerability, and (2) denial-of-service attacks do not last forever, i.e., there is some unknown point in the execution after which all messages are delivered within some constant time d. This is a rather weak synchrony assumption that is likely to hold true in any real system provided faults are eventually repaired.

Another advantage of the CLBFT algorithm is that it improves the security of the system by recovering replicas proactively. Moreover, it replaces expensive public key cryptographic operations with message authentication codes (MACs), which makes the algorithm an order of magnitude faster, and thus more practical. The CLBFT replication library has been used to implement the first Byzantine-fault-tolerant NFS file system, BFS [9]. The performance results show that BFS performs 2% faster to 24% slower than production implementations of the NFS protocol that are not replicated. These results indicate that the CLBFT replication library can be used to build practical systems that tolerate Byzantine faults.

Given these properties, we consider the CLBFT to be a practical and suitable choice for implementing a Byzantine-fault-tolerant domain name system.

Chapter 5

A Byzantine-fault-tolerant DNS with Session Key

5.1 Design Criteria

In section 3, we identified the major security problems associated with current DNS infrastructure and its security extension (DNSSEC) proposed by the Internet Engineering Task Force (IETF). As a solution to these problems, we now propose a secure and scalable Byzantine-fault-tolerant domain name system. First, we describe the design criteria the proposed system should meet. The motivations are to overcome problems associated with current DNS and its security extension on one hand, and to provide a scalable, robust infrastructure that ensures availability without assuming any specific nature of malicious attacks on the other.

The two major design criteria for the proposed system are:

- Safety, and
- Performance

We want to implement a secure domain name system, which tolerates failures and attacks both at the servers and during data communication. The system should provide a secure communication mechanism to detect whether data have been corrupted during communication over the Internet. Also, the system should tolerate server failures due to both benign faults (e.g., a server crash) and malicious attacks by hackers. Furthermore, the proposed system should support a secure dynamic update mechanism, which includes a user authentication and authorization scheme, and a mechanism to make recording of new information secure and immediately available.

Moreover, the performance of the proposed domain name system should be comparable, if not better, to what is available right now. The system should not trade performance for security and robustness to the extent that there is noticeable performance degradation.

We now discuss the design criteria for the proposed Byzantine-fault-tolerant domain name system in detail.

5.1.1 Safety

Data Integrity and Secure Communication

The proposed system should provide means for data integrity authentication, source authentication, and authorization. Using cryptographic techniques, servers should authenticate the data they send to clients, and clients should be able to verify the integrity of the data they receive.

Byzantine Fault Tolerance

Unlike the DNS security extension, the proposed system should not assume that the server private keys are not stolen, or the servers are not corrupted themselves. The system should tolerate server failures due to both benign faults, and malicious attacks. In fact, most fault tolerant algorithms assume fail-stop behavior of faulty nodes, i.e., a node stops functioning in case of a failure [34]. These algorithms are usually designed to handle unpredictable processor crashes. However, with malicious attacks on the Internet getting increasingly common, it is no longer reasonable to design a system that makes assumptions about specific behavior exhibited by faulty nodes. To ensure robustness, the system should be Byzantine fault tolerant [22], i.e., resistant to arbitrary misbehavior of faulty components [6]. However, the system should be realistic and practically realizable for an environment like the Internet.

Consistency

The system needs to ensure consistency among all the non-faulty name servers of a zone. No data update should be visible to the resolver until all the non-faulty name servers of a zone commit to that update. For a valid and authenticated update request, all the non-faulty name servers should execute the same update operations in the same order; otherwise they all should reject the request. No inconsistency in a faulty name server should be visible to the client, i.e., a client should not be confused by different versions of authenticated data at any time.

Small Window of Freshness Attack

One major problem with the DNS security extension (DNSSEC) is that the system depends on a pre-computed authentication of the data, which is usually associated with a long time to live (TTL) value to reduce the cost of signature generation. This makes the system vulnerable to the freshness attack, which occurs when an authenticated message of a previous run is recorded by an attacker and replayed as the reply to a later run. To alleviate the problem, the newly proposed system should provide per-query authentication of the data. This will significantly reduce the window of vulnerability.

Reliability and Availability

The proposed system should be highly reliable, i.e., it should ensure that the system state is not lost or corrupted when some node fails. Reliability also ensures that the effect of an operation on some data is visible to all who access the data once the operation completes, regardless of any failure that might have taken place. In addition to reliability, the system should provide availability, i.e., the service should be available even in the presence of some failure.

5.1.2 Performance

Scalability

For an exponentially growing environment like the Internet, it is of utmost importance that the proposed system be scalable. If the system does not scale well, it is unlikely that it will be adopted to provide a ubiquitous service like the domain name system, which is used by virtually all internetworking services. Especially, the system should take advantage of a smart caching scheme since caching substantially improves the performance of the domain name system.

Offload the Root Name Servers

The data in appendix A indicate that the current load on the thirteen globally distributed root name servers is too high, which significantly affects the performance of DNS. It is of utmost importance to design a system that realistically offloads the root name servers, thus ensuring better scaling and performance of the domain name system.

5.2 SBFTDNS System Architecture

We now describe the system architecture of the proposed scalable Byzantine-fault-tolerant domain name system (SBFTDNS). This system maintains the current DNS name space structure and administration. However, to provide Byzantine fault tolerance, there are 3f+1 tightly coupled replicas per name server in each zone. In this way, the system provides correct domain name service in the presence of up to f faulty replicas. Typically, f=1 to tolerate one faulty replica.

Servers

3f+1 tightly coupled replicas compose the name-server side of SBFTDNS. Each replica maintains a copy of the RR database with identical initial state. Unlike current DNS, in which there is a single key shared by an entire zone to authenticate data, each replica in SBFTDNS has its own public-private key pair. While the public key is stored and authenticated in its parent zone, or statically configured in a resolver, the private key is stored in a secure coprocessor attached to the replica and cannot be stolen. This permits real-time dynamic update without allowing a security breach, as the intruder needs to compromise more than f replicas before being able to corrupt the data.

Clients

Each client is statically configured with 3f+1 public keys that belong to the 3f+1 replicas for the name server of one trusted zone, e.g., the root zone. Clients can also learn the public keys of the replicas of a name server by constructing an authentication chain, analogous to the scheme used by the DNS security extension as described in section 3.2.2. A client obtains name service from a SBFTDNS server using the CLBFT protocol described in chapter 4.

Table 4-1 lists some of the differences between the DNS security extension (DNSSEC) and SBFTDNS.

Feature	DNSSEC	SBFTDNS
Number of faults tolerated	0	f
Number of zone replicas/name server	≥ 2	3f+1
Number of zone authentication keys	1	3f+1
Number of replicas involved in a request	1	3f+1
Real time dynamic update	Insecure	Secure

Table 4-1: Comparison of DNSSEC and SBFTDNS features

However, the SBFTDNS system architecture raises concerns about two issues. The first concerns key management, and the second concerns data caching. We now discuss these issues in detail.

5.3 Key Management

For a client to securely communicate with a server, first it needs the public key of the server. This can be easily obtained, e.g., by statically configuring each client with the public keys of the root servers, and then constructing an authentication chain to get the public key of the desired server.

However, for the client to actually communicate with the server using the CLBFT protocol, there must also be a session key established between the client and the server. This allows the use of symmetric key cryptography, which is good for a number of reasons. For example:

- Session keys are much shorter than public keys, and therefore, symmetric key digital signatures are much faster to generate and verify than public key digital signatures.

- With the session key mechanism, there is no need to generate and verify long and expensive SIG RRs that are used in DNSSEC to provide data authentication. Therefore, the reply packets are significantly smaller than those in DNSSEC and the verification cost is lower.

- In DNSSEC, some queries (e.g., the NS RR query) require a large number of SIG RRs per reply. For example, in a zone with 2 name servers (every zone has a minimum of two name servers associated with it), a security aware resolver needs to verify at least 4 SIG RR for an NS (name server) RR query where the reply includes one SIG RR for the NS RRs, one SIG RR for the zone KEY, and one SIG RR for each name server's A (address) RR (2 in total). The verification of such a large number of SIG RRs increases the elapsed time or latency of the query resolution process. The latency, however, does not increase significantly in a session key based DNS, because a resolver only needs to verify one MAC for each reply, regardless of how many RRs are included.

- Additionally, with the session key mechanism, the need to cache KEY RRs and perform the expensive KEY-SIG chain verification process is avoided.

- There is also the additional advantage of using one secret key rather than different public keys as suggested by RFC 2137 [13] for authenticating the dynamic update requests and the results of a query.

- Since the session key mechanism can significantly improve the performance of secure DNS, it enables BFTDNS to provide per-query data authentication, thereby ensuring more up-to-date data, and providing a secure, robust, and efficient domain name system.

A session key can be established between a client and a replica once the public key is obtained. Each communicating pair of client and replica shares a secret session key; this key is used for communication in both directions. The client refreshes the key periodically, using the *new-key* message. If a client neglects to do this within some system-defined period, a replica discards its current key for that client, which forces the client to refresh the key.

However, to avoid the exchange of messages to obtain the public keys, and the expensive public-key cryptographic operations to establish session keys, it is desirable to cache both public and session keys. But this raises concerns about system scalability since every client-server pair needs to share a session key to protect the communication between them from spoofing attacks. Key management may get unwieldy as the number of shared secret keys increases quadratically with the number of nodes in the system. According to a recent Internet domain survey done by the Network Wizard and published by the Internet Software Consortium [19], the total number of hosts in the Internet in July 2000 was 93,047,785 and this number was expected to cross 100M by November 2000. If we assume that every 100 hosts share a proxy name server, there are about a million proxy name servers in the Internet. Without some hierarchical organization, any of these 1M proxy name servers can theoretically communicate with any other proxy name server and therefore needs to establish a million session keys. If each session key consumes 16 bytes of memory, then a proxy name server needs 16 Mbytes of memory to cache all the session keys at the same time. If all the proxy name servers are Byzantine fault tolerant and therefore consist of 3f+1 tightly coupled replicas, resource consumption will go even higher, which may not be affordable for many small or medium size zones. In addition, to guarantee security, we refresh session keys very often in BFT. Even if we assume the average TTL of a session key is 10 days, then on average one out of a million session keys expires every second[1] (86400*10/1000000 = 0.864 sec). This incurs a significant workload on the name server as the process of establishing a session key between two entities involves expensive PPK operations. The problem of scalability, therefore, is an important one.

[1] Assume Poisson arrival of session key expiration times with arrival rate $\lambda = 1/10$days
With N=1000000, the aggregate rate = $N\lambda$ = 100000/day.
The average inter-arrival time = $1/N\lambda$ = 1/100000 day = 86400/100000 sec = 0.864 sec

To keep the problem of scalability in a Byzantine-fault-tolerant DNS under control, we want a system organization that minimizes the number of session keys. However, the number of session keys that need to be maintained between different DNS entities is tightly coupled with DNS query resolution scheme because the query resolution scheme dictates who gets to contact whom, and therefore who needs to establish a session key with whom. We now introduce various query resolution approaches, discuss their advantages and disadvantages, especially with regard to its potential to minimize the number of session keys.

Approach 1: Fully Iterative

In this approach, the resolver takes full responsibility on resolving a query. First, it iteratively queries the local name server. If the answer is not found in the local name server database or cache, the local name server directs the resolver to the root name server. The rest of the resolution process is done in a similar iterative fashion, as demonstrated in figure 5-1.

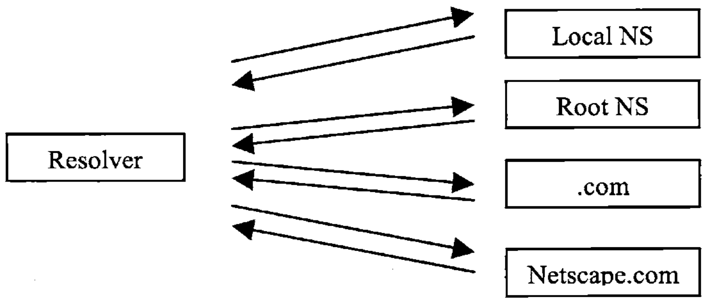


Figure 5-1: A Fully Iterative Resolution Scheme

Pros

- Being fully iterative, this scheme reduces the load on the intermediate name servers, each of which simply directs the client to the closest source of information without using its own server time querying the other name servers.
- The burden of the resolution process is placed on the resolver that issues the query, which should be fine for an efficient, caching resolver.
- In this scheme, the resolver may cache not only the final answer, but also the answers of the intermediate queries. For example, a full iterative query for the IP address of www.foo.edu.ca results in the caching of the IP addresses of www.foo.edu.ca, foo.edu.ca, edu.ca and .ca servers. Therefore, the next time the same resolver queries for the IP address of www.bar.edu.ca, it does not have to go through the full resolution path; rather, it can make a short cut and start the resolution process by directly asking the edu.ca servers. This off-loads the root name servers.
- This scheme requires a smart, possibly caching resolver, which may be useful. The results of the analysis in appendix A indicate that caching (with support for negative caching) in resolvers may suppress a lot of network traffic due to repetitive queries.

Cons

- The major problem with this design is that a name server may be contacted by any resolver on the Internet, and therefore needs to establish session keys with each of them. According to [19] there are about 100 million hosts on the Internet right now. Assuming each of them is capable of issuing a DNS query, this renders a huge number of possible session keys for each name server even though not all of these session keys need to be cached at the same time.
- Furthermore, this scheme calls for an efficient, caching resolver, which is uncommon in DNS today. Common DNS implementations include simple *stub* resolvers that do not have any caches. Such resolvers typically make recursive queries to a caching proxy name server (usually the local name server), which eventually returns the final answer or an error to the resolver.
- Also, the caching of the RRs in the resolver does not achieve much, because both the final and the intermediate results are cached in the resolver that issues the query, not in the intermediate name servers. Only related queries later issued from the same resolver can take advantage of the caching. Queries from other resolvers will have to go through the entire resolution mechanism.

Approach 2: Hybrid Scheme

In this approach, the local name server acts as a proxy server. A resolver makes a recursive query to the local name server, which, in turn, makes an iterative query to the root name server. The rest of the resolution process is carried out in a similar iterative fashion, as depicted in the figure. Once the result of the query is obtained, the local name server returns the result to the resolver. This is the most common resolution scheme in the Internet today.

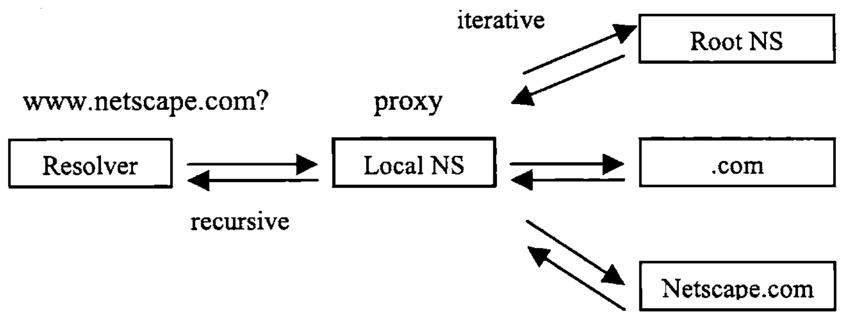


Figure 5-2: A Hybrid of Iterative and Recursive Resolution Schemes

Pros

- Session keys between the resolvers and their query proxies are easy to cache, because the number of resolvers served by the local name server is relatively small.
- This scheme fits the most common DNS resolution approach today. It allows for a *not-so-intelligent* stub resolver and provides an intermediary proxy server that may ensure security, administrative control, and caching service.

- As intermediate results of the query can be cached in the proxy server, all the hosts that share the same proxy name server may benefit from the caching of the final and the intermediate answers of a query.

Cons

- The major problem of this most commonly adopted approach is that all the queries that cannot be resolved by the local name server using its own database and cache are directed to the root name servers. This puts a huge load on the thirteen globally distributed root name servers. As we discuss in appendix A, 1/3 of the wide-area DNS traffic that traversed the NSFnet in 1992 was destined to the seven root name servers [11], while we observed almost 19% of DNS queries going out of MIT's Laboratory for Computer Science (LCS) in January 2000 contacted a root name server with a significant impact on the performance. Given the exponential growth of the Internet, this is inefficient and non-scalable, as the root name servers become the bottleneck of the system.

Approach 3: Fully Recursive

In this approach, queries are handled fully recursively. A resolver makes a recursive query to the local name server. If the query cannot be resolved locally, the local name server makes a recursive query to the root name server. From then on the process continues in a similar recursive fashion until the final result is found.

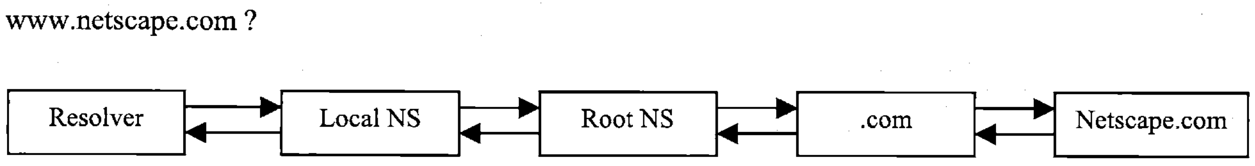


Figure 5-3: A Fully Recursive Resolution Scheme

Pros

- In this scheme, a name server only needs to cache the session keys for its superzone and subzone. Thus, small zones only need to cache a few session keys, since the number of their superzones is small and they generally have no or just a few subzones.

Cons

- The biggest disadvantage of this scheme is that the workload on the top-level name servers, especially the root servers, increases significantly. Every query that cannot be resolved locally reaches the root name servers. Furthermore, root servers can no longer simply refer the querier to some other name server. Rather, they are now obliged to resolve the query, which makes this scheme quite impractical.
- Any local name server may contact the root name server. Therefore, the possible number of session keys at the root level is large. The number is also large in the top-level zones, for example, the .com zone has nearly two million subzones. Without some load distribution

scheme, this approach may end up requiring huge number of session keys to be maintained at the top level, even though the number is a lot lower than what is required for the fully iterative scheme.

- Also, in this scheme, the local name servers cache only the final results, not the intermediate results. To take full advantage of caching, intermediate results could be passed back to the local name server, but that would increase DNS traffic in the Internet substantially.

Approach 4: Fully Recursive with Intermediate Hops

In this approach, the resolution mechanism is recursive all the way with additional intermediate hops from the local name server to the root name server. For example, let us suppose a resolver in the lcs.mit.edu domain issues a query for the IP address of www.netscape.com. The query first reaches the local lcs.mit.edu name server. If it cannot be resolved locally, then the lcs.mit.edu server contacts an mit.edu server. If not resolved, mit.edu server contacts .edu, which, in turn, contacts the root name server and the rest of the query is resolved in a fashion similar to the one described in approach 3. It should be noted that both the .com and the .edu zones are maintained by the root name servers. The figure is drawn in a way to represent the hierarchy of resolution.

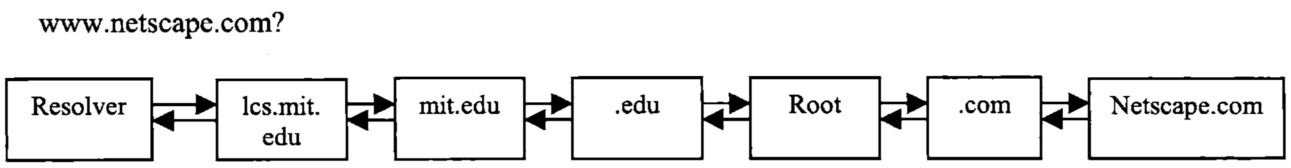


Figure 5-4: A Fully Recursive Resolution Scheme with Intermediate Hops

Pros

- This approach reduces the load on the root name servers. All queries that belong to the same top-level domain as the querier are resolved without contacting the root name servers. This may off-load root name servers substantially.
- As in approach 3, a name server only needs to cache the session keys for its superzone and subzone. It has the additional advantage that a root name server is not contacted by just any local name server, but only the name servers of its subzones. This reduces the number of session keys to be maintained by the root name servers.

Cons

- This scheme increases the number of hops between the local name server and the root name servers.
- Root name servers are still subject to recursive queries, which increase the amount of load on the root name servers.
- Current DNS does not support full recursion, at least not at the root name servers.
- Most of all, the advantages of the scheme may not be that significant as the current organization of DNS name space is mostly flat with 2 to 3 levels of hierarchy. The cost of recursion may very well outweigh the gain achieved by the small number of intermediate hops.

5.3.1 Design of Choice: A Hybrid Resolution Approach with Intermediate Level Hierarchy

Based on the analysis of the advantages and the disadvantages of various resolution schemes, we propose a resolution approach that combines both the iterative and the recursive resolution techniques, thereby capturing the best of both worlds. To minimize the number of session keys and take full advantage of caching, the system is enhanced with intermediate levels of hierarchy.

In this approach, a resolver (which may be a stub resolver with no caching) places a recursive query to the local name server. To offload the root name servers, we propose multilevel recursive hops from the local name server to a level right below the root name server. From the level right below the root name server and on, the query is resolved in an iterative fashion. Figure 5-5 describes a generalized version of the scheme with n=2 intermediate levels between the local name server and the root name servers, where n is arbitrary and can be chosen appropriately.

Note also that an nth level server can handle the load from the (n-1)st level servers that communicate with the rest of DNS through it because we do data caching at each level (see section 5.4). Therefore an (n-1)st level server communicates with an nth level server only occasionally.

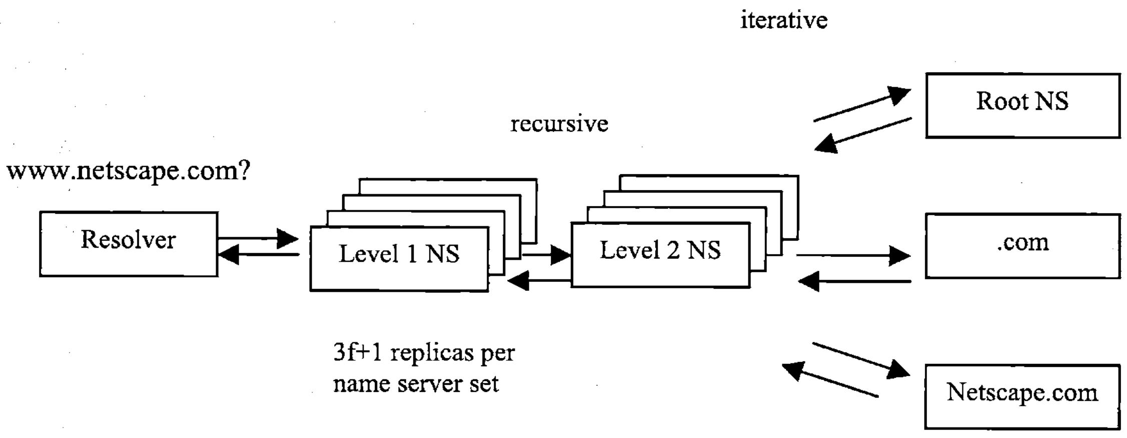


Figure 5-5: A Hybrid Resolution Scheme with n=2 Intermediate Levels of Hierarchy

As we discussed before, more than 90% of current DNS name space is flat with only 2 to 3 levels of depth. One might find a 4 to 5 level deep organization of DNS name space only in an academic or a research organization. Therefore, the intermediate levels we propose may or may not be strictly related to the domain name space hierarchy. One might argue that arranging the hosts and name servers strictly according to the name space hierarchy has the advantage that all queries whose answers belong to the same top level domain as the resolver can be resolved without contacting the root name servers. This might yield a significant gain in terms of load reduction on root servers, because a substantial portion of DNS queries is confined within the same top-level domain. However, offloading the root is possible with any intermediate level hierarchy even when the intermediate levels are not dictated by DNS name space hierarchy. Moreover, for huge zones like the .com, which has 32 million hosts, one needs to deploy some load distribution scheme anyway. For example, queries from specific servers may need to be configured to contact some specific set of .com servers to distribute the load on the zone. With the exponential growth of the Internet, we believe, such schemes need to be exploited anyway to ensure the scalability of the domain name system. Therefore, how to aggregate the hosts and the name servers into intermediate levels to create an efficient hierarchical structure may entirely be an administrative decision. The point we want to emphasize is that an efficient domain name system should make extensive use of sharing caches among clients, since sharing caches, as indicated by the data in appendix A, significantly increases DNS cache hit rates.

The recursive part of the proposed resolution scheme, together with the intermediate levels of hierarchy, significantly reduces the number of session keys a name server needs to maintain. The iterative part, on the other hand, substantially offloads the root name servers, which is of utmost importance for the scalability and performance of DNS. The scheme makes full use of caching the intermediate results and using them to short-cut future related queries, which is an advantage of the iterative over the recursive resolution scheme.

In addition, the scheme can easily be employed in current DNS by taking advantage of the existing *forwarder* directive provided by the domain name system software like BIND. The addition of a forwarder statement in the name daemon boot file makes a local name server recursively direct any locally unresolved query to the forwarder name server. Because of the rich cache of information built at the forwarder, there is a high probability that the forwarder can answer a given query in a remote domain from its cache. In addition to offloading the root name server, the scheme holds the potential for limiting off-site DNS traffic to the bare minimum, which might prove useful when the network connection is pay-per-packet or the network connection is a slow link with a high delay.

The proposed resolution scheme thus combines the advantages of both the iterative and the recursive approach. Even though it increases the number of hops from the local name server to the root name servers, the associated cost is low. The advantage of off-loading the root servers by making full use of the intermediate level hierarchy substantially outweighs the disadvantages of the scheme. Moreover, the hierarchical organization keeps the number of session keys low, thereby ensuring a scalable and secure domain name system.

5.3.2 Replicated Clients

This preferred organization ends up with a Byzantine-fault-tolerant server, consisting of 3f+1 replicas, acting as a client to another name server, i.e., a group of replicas acting as a client. And this could easily lead to very large number of messages, e.g., if each replica in the client communicated with each replica in the server. Let us consider a system with two levels of Byzantine tolerant replicated servers as illustrated in figure 5-6. Each server consists of four (3f+1 = 4) replicas to tolerate up to one failure. Suppose, a BFT client wants to resolve the IP address of some host. It sends out four query messages to the four replicated servers in level 1. Suppose the query could not be resolved by the level-1 replicas. Now each of the replicas can act as a Byzantine-fault-tolerant client and therefore independently send four requests to each of the 2nd level replicas. In this way, the number of messages increases geometrically with the number of replicas per set as the level of hierarchy increases. Even in the case that the query can be resolved by the 2nd level replicas, it would require 40 ($4*2 + 4^2*2$) messages, which incurs a huge amount of network overhead.

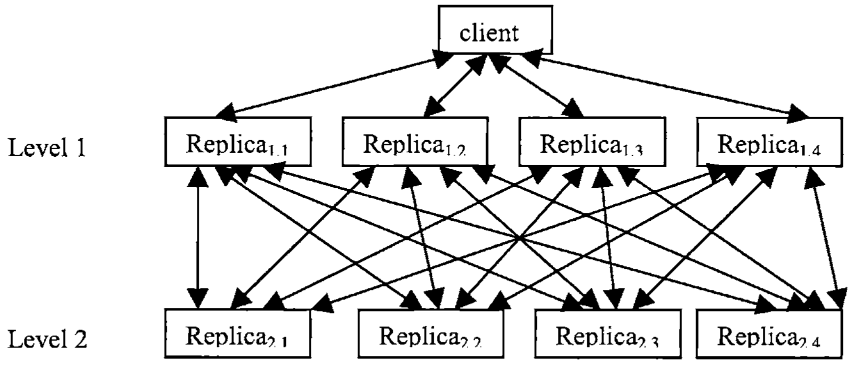


Figure 5-6: Increase in the Number of Messages Across Levels of Hierarchy

We propose a scalable solution to this problem by having the primary of the client-group perform the query on behalf of the other replicas in its group, as illustrated in figure 5-7. After resolving the query, primary sends the reply to the backups. After verification, the replicas send the reply back to the original client as usual. Of course, the primary may be faulty. Therefore, the replies sent by the 2nd level of replicas are constructed in a special way so that the replicas in level 1 can verify the integrity and authenticity of the reply independent of whoever performed the query on their behalf. The mechanism is described in detail in section 6.4.6.

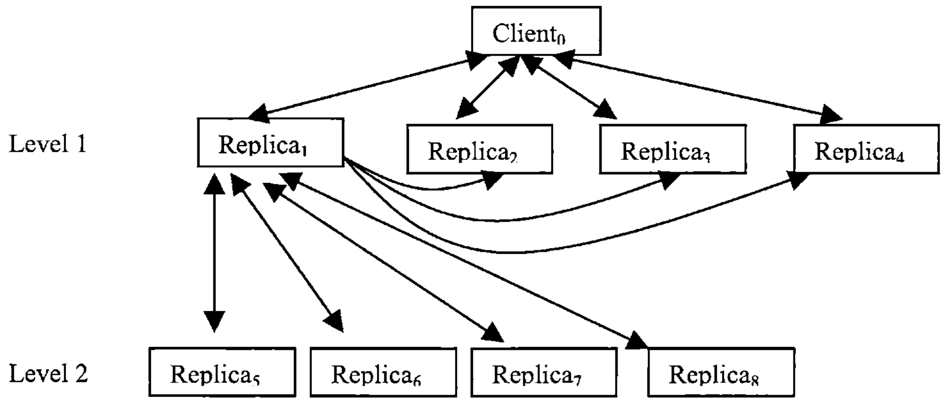


Figure 5-7: A Scheme to Support Replicated Clients with Reduced Number of Messages

Using this scheme, we limit the number of message that each replica of one set receives from the replicas of another set to 1, which substantially reduces the load on the servers. The maximum number of messages that one set of replica sends to another set, therefore, is limited to n rather than n^{level}, where n is the number of replicas per set and level indicates the level of hierarchy. Even though the scheme increases the size of the specially constructed reply, it significantly reduces the network traffic and overhead (in terms of headers) that would otherwise be needed for replicated clients. In section

7.4.3, we present the performance evaluation of the proposed scalable Byzantine-fault-tolerant domain name system with and without the support for reducing the number of messages across levels. The results show that even with a two-level hierarchy, the reduction scheme almost halves the number of messages resulting from a read-only request, and yields a 67% reduction in the total number of reply bytes traversing the network.

5.4 Caching Mechanism

For the system to perform well, we also need an efficient data caching system. Caching is tightly coupled with the performance and security of a system. The domain name system, being a very large-scale distributed system, makes extensive use of resource record caching features. Most of the operations in DNS are read-only, i.e., they do not modify the state of the system. Moreover, data changes are very infrequent in DNS. Stale information therefore occurs infrequently in the domain name system. This makes caching a good technique to improve performance in DNS. In fact, caching has been an integral part of DNS since its description in RFCs 1034 and 1035 [25][26].

In general, DNS caching exploits the idea of temporal locality since it is often the case that an item referenced recently is likely to be referenced again in the near future. Without caching, all non-local DNS query would have to contact the root name servers and go through the full resolution path. Caching the results of a query, e.g., an A RR, in the local name server allows the server to provide the answer to a later reference of the same query immediately from its local cache without having to go through the full resolution process. Even caching NS (name server) RRs has important benefits, as indicated by the results we describe in appendix A. When a name server finds the NS record for the target zone of a query in its cache, it can bypass the root name server and contact the name server of the target zone via a "short path." Caching thus offloads the root name servers and reduces unnecessary network traffic. On the client side, caching reduces the long latencies due to multiple network round-trips before getting an answer to a query. Caching thus benefits both servers and clients of the domain name system.

However, caching raises concerns about consistency and staleness of data. We discuss this problem in detail in the following section.

5.4.1 Security and Cache Consistency

The original design of DNS sacrificed consistency in favor of reduced access time. For a secure domain name system, however, stale information may no longer be harmless since it may involve some security-critical information, e.g., a compromised private key. Stale information opens a window for a freshness or replay attack, and thereby poses a security threat to the system. The issue of cache consistency, therefore, becomes critical for a security aware DNS.

DNS typically uses TTLs (Time To Live) to control cache consistency [25]. Resource record TTLs are typically assigned by the administrator of the zone the RR belongs to. When an object in the cache hits zero TTL, it is evicted from the cache. The DNS security extension (DNSSEC) [17] proposes the use of two TTL values: original TTL and current TTL. The original TTL value of the RR is protected by a digital signature while the current TTL field is not. Under this scheme, unscrupulous servers may manipulate the current TTL, but a security aware resolver will bound the remaining time-to-live value of the object at the original TTL value if some discrepancy is found. Also, a security aware server in DNSSEC must not consider SIG RRs to authenticate anything before the signature inception time or after the signature expiration time. Therefore, when a secure server caches authenticated data, if the TTL would expire at a time further in the future than the signature expiration time, the server should trim the TTL in the cache entry not to extend beyond the signature expiration time. Within these constraints, servers continue to follow DNS TTL aging. In general, the TTL on an RR in DNSSEC is:

$$\text{TTL} = \text{Min}(\text{SigExpTime}, \max(\text{zoneMinTTL}, \min(\text{originalTTL}, \text{currentTTL})))$$

DNSSEC proposes to set the signature expiration times far enough in the future so that it is quite certain that new signatures can be generated before the old ones expire [17]. However, setting the expiration time too far into the future could mean a long time to flush bad data, thereby increasing the window of replay attack.

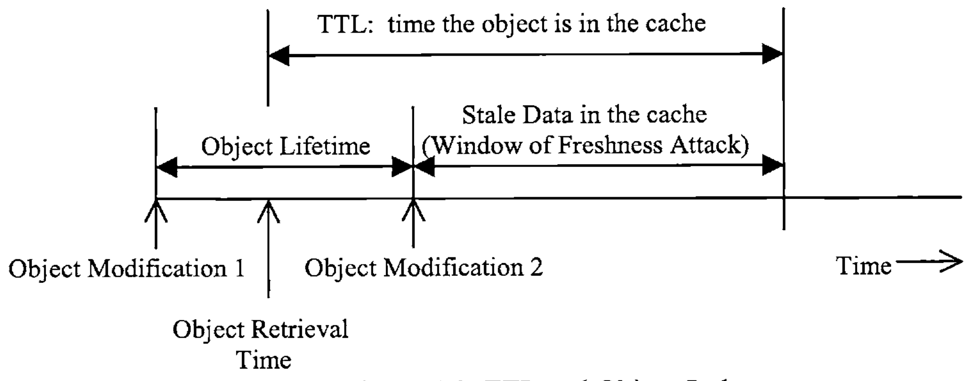


Figure 5-8: TTL and Object Staleness

Figure 5-8 illustrates the TTLs, object staleness, and the window of freshness attack for an object cached in a client. The TTL indicates the time the object is cached in the client. As shown in figure 5-8, the object is modified in the server after it was retrieved and cached by the client. The time between the second modification and the end of the TTL is the window of freshness attack for this object since the client has a stale copy of the object in its cache during this period.

In a Byzantine-fault-tolerant DNS with 3f+1 tightly coupled replicas per server, manipulating the TTL in one replica does not cause any major problem, as consensus among non-faulty replicas ensures the correct TTL value. In SBFTDNS with a session key, cached RRs, when sent to the clients, are authenticated in the same way as the normal RRs in the database, i.e., using message authentication codes generated using session keys of the communicating nodes. Since SBFTDNS does not use SIG RRs, staleness due to signature expiration time does not arise.

It should be noted that it is not possible to entirely replace the TTL mechanism. The TTL is primarily a database consistency mechanism and non-security aware servers that depend on the TTL must still be supported. The TTL, however, is a very poor heuristic to provide cache consistency. Often times, TTLs are determined by the cache owners rather than the owner of the object, which is a problem since the cache administrator hardly knows the details of any individual object in the cache. Especially for security critical systems, where stale data can be dangerous, administrators will be inclined to use very short TTLs so that changes take effect rapidly. However, in reality, changes might be very rare and may not need timeliness. A short TTL in such cases may unnecessarily place significant load on the network links and servers. On the other hand, a long TTL increases the window of freshness attack. If some critical change has been done to data with a long TTL, there isn't much that can be done to stop the distribution of the stale data. To solve both the problems, we propose invalidation as a consistency mechanism on top of a hierarchical caching system.

5.4.2 Proposition: A Hierarchical Invalidation Protocol using Leases

Traditionally, designers of large distributed systems have been willing to live with some degree of cache inconsistency to reduce server hot spots, network traffic, and data retrieval latency. However, as the Internet rapidly evolves to provide more and more mission critical services, the issue of cache consistency becomes critical. To minimize the window of freshness attack, data updates should propagate quickly from the source to the sink. For a large-scale system like DNS, it is important that the mechanism for cache consistency be scalable as well. Since updates are rare, it makes sense to use the intrinsic hierarchy of the domain name space and build a hierarchical cache with an invalidation mechanism to ensure data coherence.

The first step is to determine who initiates the invalidation process – the client or the server.

- **A Client-driven** invalidation mechanism is a reactive process. Every time a cached data item is referred to, the client checks with the server to determine whether the cached copy is valid. The advantage is that this is a stateless protocol, i.e., the servers do not need to maintain any state information about the clients who have cached copies of the data. When the cost of validation is significantly lower than fetching the data, a client-driven process reduces the server load and improves the client performance. However, the problem with this scheme is that the client does not know when data change at the server, and therefore it must make many unnecessary checks, which can degrade system performance. A client-driven process essentially defeats the purpose of caching, and may prove impractical for a large-scale system because of the large amount of traffic it generates and the load it places on the server. The amount of overhead traffic here is proportional to the client hit rate.

- **A Server-driven** invalidation mechanism, on the other hand, is a proactive process. When a data item changes, the server notifies clients that have cached copies. For a large-scale system like DNS, where data updates are very rare, a server-driven invalidation mechanism may perform better than a client-driven invalidation scheme. The problem, however, is that the server now needs to maintain state for the cached objects. The performance of the system degrades if the server's callback state becomes excessively large. Also, in case of a client failure, the server may have to wait for an unbounded time for the client's acknowledgement of the invalidation message. Furthermore, a server-driven scheme cannot guarantee complete consistency because of the delay in invalidations arriving at clients.

Leases

We propose a hybrid of the server-driven and the client-driven invalidation schemes using leases [18]. To limit the polling for data invalidations, the servers give an invalidation contract or lease to the caching clients, as in the Andrew File System (AFS) [21]. The lease is for a certain amount of time within which the server guarantees to notify the caching client of the data updates, as illustrated in figure 5-9. Once the lease is obtained, the system turns into a reactive or a client-driven system. It is now the responsibility of the client to keep extending the lease if it still needs it. The lease period is short to limit the time during which delay in arrival of invalidation messages can cause an inconsistency at the client cache.

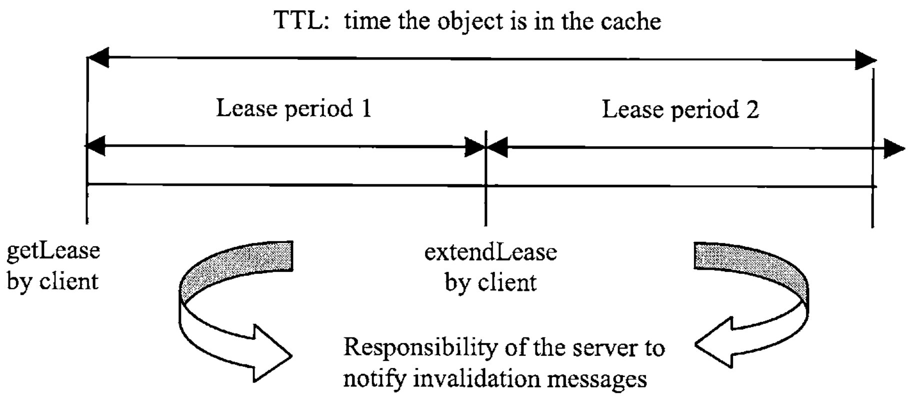

TTL: time the object is in the cache
Lease period 1
Lease period 2
getLease by client
extendLease by client
Responsibility of the server to notify invalidation messages

Figure 5-9: A Hybrid of Client-driven and Server-driven Invalidation schemes with Leases

We propose a lease-per-client scheme. With this scheme, the amount of state that the server must store per client is minimal. This scheme has the additional advantage that a lease covers all the data a client has cached from a server. Therefore, when the client requests a new lease, e.g., because of a query affecting some object in its cache, it gets an extension on all other objects belonging to that server that are in its cache. Since the server does not track the objects stored at the client, it may send unnecessary invalidations. We believe this is not a problem, however, because updates in DNS are very rare.

Mechanism to Obtain a Lease

We now describe how a client can obtain and extend a lease from a server.

There are two requests for getting leases. The first is *getLease*. This can only be called when a client is currently caching no objects from the server, e.g., at the first reference to an object from this server.

The second request is *extendLease*. This is called when a client already has a lease; it asks for the lease to be extended. If the server still holds the lease for the client, it will extend the lease; otherwise it refuses the request. The client extends its lease periodically, e.g., if a lease is good for five minutes, a client might refresh after three minutes. The goal is to not do this very often, but to do it far enough in advance so that the *extendLease* request is never refused. Since leases are only refreshed occasionally, these requests add negligible overhead.

If the server decides to extend the lease requested by the client, it sends a list of pending invalidations (i.e., ones for which this client hasn't replied to the invalidation yet) in the response to the *extendLease* request. This list of pending invalidations is needed since the reply to the *extendLease* message implies that all objects that arrived in the client cache prior to the extendLease request and that are not explicitly listed in the reply are valid.

If the sever refuses the *extendLease* request (e.g. the server does not hold a lease for the client), the client must discard all cached objects that belong to that server. It is necessary to discard all objects because there can be missing invalidations that the client was not notified of since the server thought it had no lease for this client.

Lease requests are modifications and therefore they will be totally ordered with respect to all other modifications at the server. We also propose that lease requests from a client be ordered with respect to queries from that client. This isn't strictly necessary but it will guarantee that there are no spurious discards from the client cache, e.g., where the client receives an invalidation for object X in a response to a lease request, but in fact it has already discarded the invalid version of X and refetched it. To order lease requests relative to queries, clients cannot have outstanding reads for that server when it makes a lease request.

Finally, to speed things up, we can combine the lease requests with queries. In this case a *getLease* request takes a query as an argument; in return the client gets the result and the lease. *ExtendLease* is similar if it succeeds; if it fails, the client gets nothing.

Hierarchical Object Caching

For a lease-per-client scheme, the amount of state that the server must store per client is minimal. To further reduce the size of the server state and the load on the servers, we take advantage of the hierarchical topology of the proposed scalable BFTDNS to implement a hierarchical object-caching mechanism. The hierarchical organization of the proxy name servers limits the amount of state to be maintained in one server and reduces the load on a particular server while performing invalidations.

We can demonstrate this advantage using the following scenario illustrated in figure 5-10. If we assume that a server takes 10ms to send one invalidation notice, and invalidations are sent from left to right with no invalidation messages being lost, then it takes 120 ms for server S to complete its invalidation in scenario 1 with flat topology. With 1-level hierarchy in scenario 2, it takes 40 ms, whereas in scenario 3 with 2-level hierarchy, it takes 20 ms for S to complete sending the invalidation messages. The invalidation message reaches node 12 at the 120th ms in scenario 1, while it takes 60 ms in scenario 2 and 50 ms in scenario 3.

One might argue that for a hierarchical caching system, there is a trade off between the client response time and the server load because a request reaches the server that owns the data only after all the intermediate caches have been searched. This disadvantage, however, is amortized by the high probability that the answer is present in some intermediate cache, because of the rich cache of information built at the intermediate name servers due to the recursive resolution scheme.

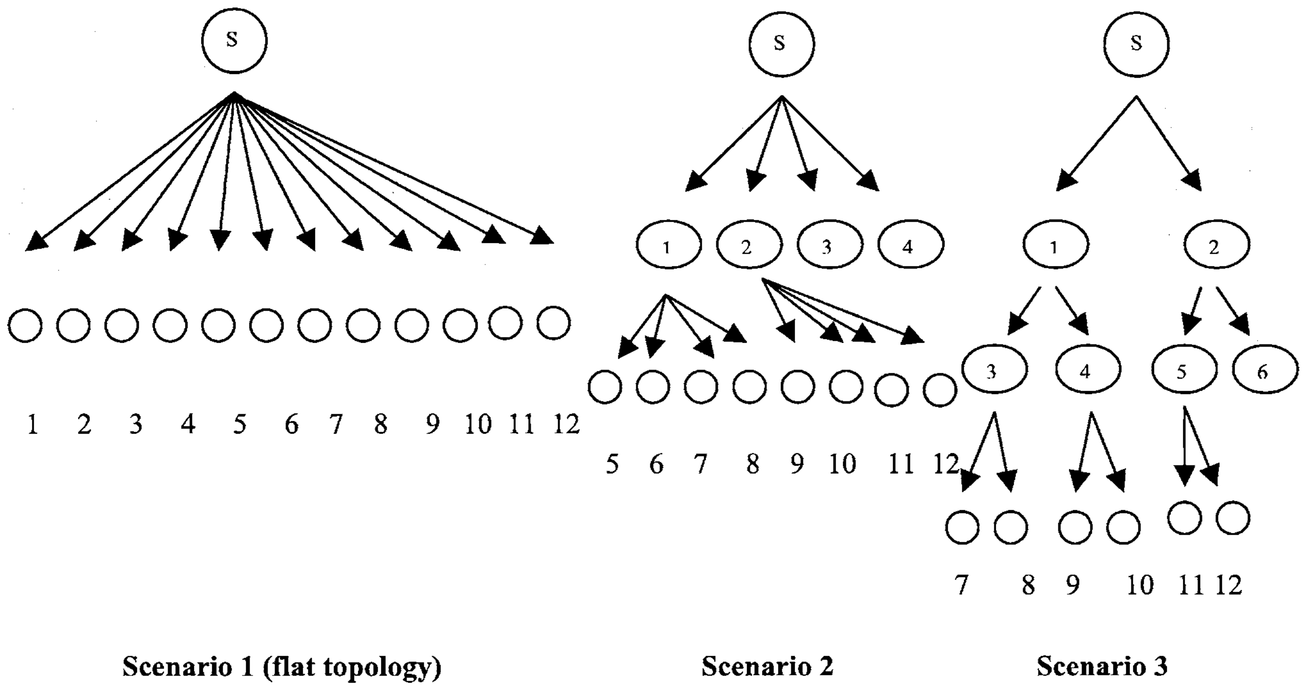


Figure 5-10: Advantage of Hierarchical Caching

5.4.3 Summary of Caching

With the exponential growth of the Internet, system performance and scalability are two major issues to be concerned about. To enhance the performance of the domain name system, we allow caching of session keys and resource records. For a security aware system, however, cache incoherence is no longer harmless. To ensure cache consistency while reducing the load on the servers due to short TTLs, we provide an invalidation mechanism with leases. To address the system scalability issue, we propose organizing the proxy name servers into intermediate levels of hierarchy creating a hierarchical organization of caches. The scheme naturally fits DNS hierarchy and provides system scalability together with reduced network bandwidth consumption, reduced access latency, and improved resiliency.

5.5 Running Operations

Now that we have the system organization in place, we can describe how operations execute in the proposed scalable Byzantine-fault-tolerant DNS with a hierarchical caching scheme.

Client places a request to the first level server using the CLBFT protocol described in section 4.3. The first level server immediately sends a reply back to the client if the answer is found in the server's local database or in its cache. However, if the request cannot be served locally, the first level server sends the request to the next level and the process continues. This is illustrated in figure 5-11. A client C queries a group of level-1 replicas for an answer that can be resolved locally by the replicas, the level-1 replicas act as servers for the original client C.

However, if the answer for the query cannot be resolved locally by the level-1 replicas, they need to act as clients for the level-2 replica set to proceed with the query resolution process. Once the results are obtained from the 2nd level replicas, the level-1 replicas authenticate the answer and send it to the original clients. Since groups can take on various roles, the system needs to be concerned about synchronization between their activities in the various roles.

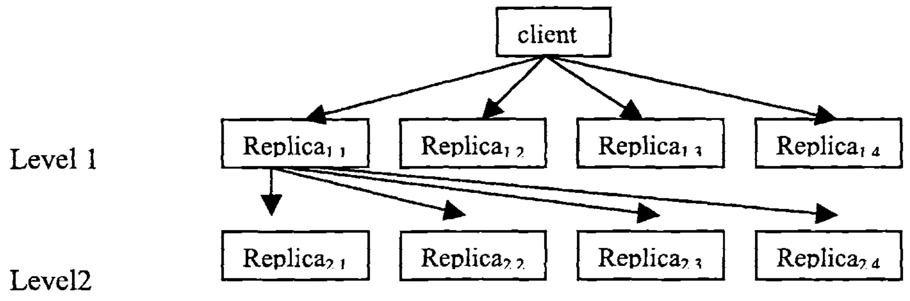


Figure 5-11: Replicated Clients in SBFTDNS

For read requests, a server at a higher level is contacted by a lower level server only if there is no cache-hit in the lower level server. Caching thus reduces both the load on the name servers and unnecessary network traffic. For update requests, however, caching does not help much. An update request is passed through until it arrives at the server (consisting of 3f+1 replicas) of a zone that owns the data being modified. The server will then send invalidations to all other servers that have active leases. Therefore, updates lead to considerable communication in the Byzantine-fault-tolerant DNS. But this is not a serious issue since updates in DNS are likely to be extremely rare.

Chapter 6

Implementation

In this chapter, we give an overview of the implementation of the proposed scalable Byzantine-fault-tolerant DNS (SBFTDNS) with session key caching mechanism. The system integrates the Castro-Liskov Byzantine-fault-tolerant replication library (CLBFT) [6] with TIS/DNSSEC [17], a beta version of the secure domain name system developed by Trusted Information Systems that partially incorporates the DNS security extensions proposed by the IETF. We first give a brief description of the implementation of the CLBFT replication library and TIS/DNSSEC. We then discuss the architecture of SBFTDNS together with the enhancement and modifications needed to support the session-key based security mechanism and the scalability of the system.

6.1 The CLBFT Replication Library

The CLBFT replication library is implemented with a very simple interface (figure 6-1), which can be used to provide Byzantine fault tolerance in any replicated service. Some components of the library run on the clients and others at the replicas.

Client:

```
int Byz_init_client (char *conf);
int Byz_invoke (Byz_req *req, Byz_rep *rep, bool read_only) ;
```

Server:

```
int Byz_init_replica (char *cnf, char *mem, int size, UC exec);
void Byz_modify (char *mod, int size);
```

Server Upcall:

```
int execute (Byz_req *req, Byz_rep *rep, int client);
```

Figure 6-1: The CLBFT Replication Library API

On the client side, the library provides a procedure *Byz_init_client* to initialize the client using a configuration file, which contains the public keys and IP addresses of the replicas. The library also provides a procedure, *Byz_invoke,* that is called to cause an operation to be executed. This procedure carries out the client side of the protocol and returns the result when enough replicas have responded.

On the server side, the system provides an initialization procedure *Byz_init_replica,* that takes as arguments a configuration file with the public keys and the IP addresses of the replicas and clients, the region of memory where the application state is stored, and a procedure to execute requests. When the system needs to execute an operation, it makes an upcall to the *execute* procedure. This procedure carries out the operation as specified for the application, using the application state. As the application performs the operation, each time it is about to modify the application state, it calls the *Byz_modify* procedure to notify the location about to be modified. This call allows efficient maintenance of checkpoints and digest computation.

6.2 TIS/DNSSEC

TIS/DNSSEC we used is the beta version 1.4 of a secure domain name system developed by the Trusted Information Systems. The reason we chose to develop SBFTDNS on top of TIS/DNSSEC, rather than writing it from scratch is that TIS/DNSSEC partially implements the DNS security extensions (DNSSEC) proposed by the IETF. At the time of this implementation, this was the only implementation of DNSSEC available, although BIND 9.0 has recently incorporated the features implemented by TIS/DNSSEC [20] and thereby declared TIS/DNSSEC to be obsolete. Also, TIS/DNSSEC provides vast compatibility as it is based on the Berkeley Internet Name Domain (BIND version 4.9.4-pl), which is estimated to be DNS software used by over 90% of the hosts in the Internet.

TIS/DNSSEC consists of a name server program called *named* (name daemon) and a resolver library, and several utility programs. The overall architecture of TIS/DNSSEC is shown in figure 6-2.

Following are some of the major modifications that TIS/DNSSEC made to BIND to support security as proposed by the IETF:

- A signature generator module has been added to the name servers. This generator reads the RRs from the zone database and generates the corresponding SIG RRs and NXT RRs using the zone private key, then adds these RRs to the database. It uses RSA PPK algorithm for the signing process.
- In addition to returning the requested RRs, the database query module also returns the corresponding SIG RRs or NXT RRs where applicable.
- The overall system is modified to properly handle newly added RR types, e.g., SIG RRs, NXT RRs etc. as proposed by the IETF to support security in DNS.

TIS/DNSSEC is a partial implementation of the DNS security extension proposed by the IETF. The beta version 1.4 of TIS/DNSSEC, which was used in our implementation, does not support dynamic update of zone RRs. Moreover, it does not provide any signature verifier at the client side, which is odd because there is no point degrading the system performance by signing the data at the server side if the authenticity of the results of a query is not verified.

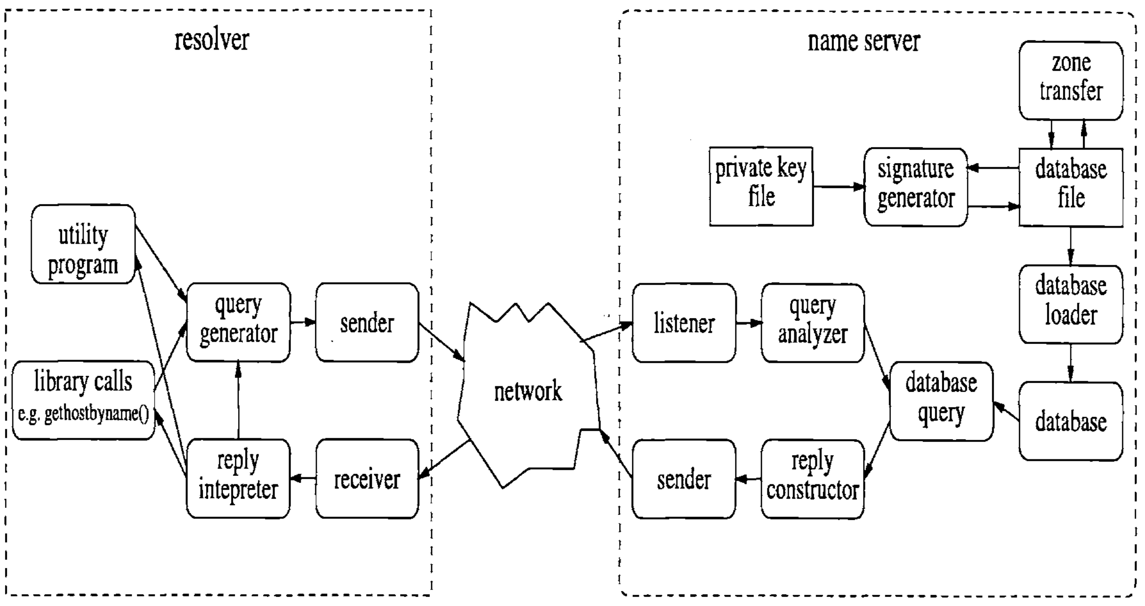


Figure 6-2: The Architecture of TIS/DNSSEC

6.3 SBFTDNS with Session Key Mechanism

The proposed scalable Byzantine-fault-tolerant DNS is built by combining TIS/DNSSEC with the CLBFT replication library with appropriate enhancement and modifications to provide the features described in section 5.1. Figure 6-3 describes the overall architecture of the scalable BFTDNS with session key caching mechanism (SBFTDNS). A detail description of the underlying architecture of BFTDNS can be found at [40]. In this section, we discuss the major modifications and enhancements that were necessary to provide the scalability of the system.

In short, the sender and receiver modules in the client side of TIS/DNSSEC have been replaced by the CLBFT replication library calls. An incoming request triggers the *execute* procedure in the server, thereby executing a read or invalidation request, depending upon the request type. The request is authenticated only for the case when the corresponding operation changes the state of the system, for example an invalidation or an update request. Since the design philosophy of DNS dictates that DNS data should be accessible by anybody, BFTDNS servers do not verify the authenticity of read-only requests. This modification in the CLBFT replication library to disable authentication check of read requests enhances system performance. Although an authentication of read requests may be useful to provide a check against flooding attacks, we, like the original DNS, are not too concerned about it.

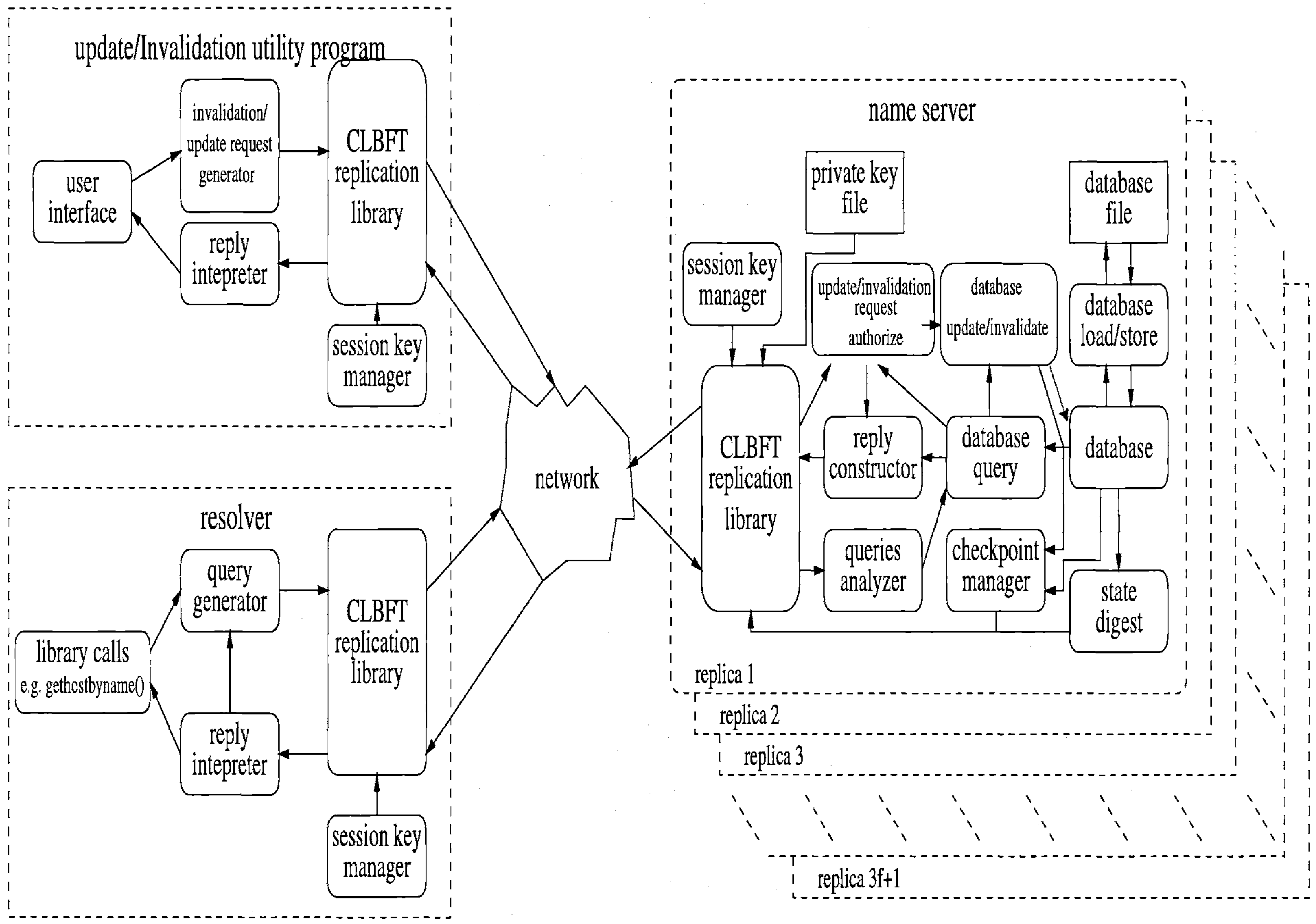


Figure 6-3: Architecture of SBFTDNS

6.4 Modifications to Provide System Scalability

In this section we discuss the major components added or modified to specifically address the scalability issues.

6.4.1 Replicated Clients

In the original CLBFT replication library, a node can function either as a Byzantine-fault-tolerant client or as a replica, but not as both. However, to support the hierarchy of name servers, replicas often need to function as clients, as discussed in section 5.5. In figure 6-4, replicas of level 1 act as servers for client C, but as clients for the level-2 replicas when the replicas of level 1 cannot serve some request using their own cache or database. To support replicated-clients, the CLBFT replication library was modified as necessary so that a node can change its role from being a client to a replica and vice-versa. The original session key mechanism between clients and replicas was extended to authenticate communication between replicas.

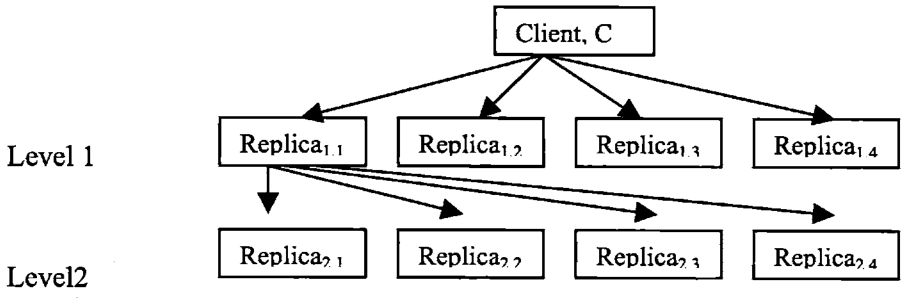


Figure 6-4: Hierarchical Organization of Name Servers

6.4.2 RR Caching

The support of caching is crucial for system performance and scalability. However, for a replicated system like the Byzantine-fault-tolerant DNS, the support of RR caching with TTL (time-to-live) involves some complications in establishing consistency among replica states. Let us consider the example where a Byzantine-fault-tolerant client makes a query for the IP address of netscape.com to a replicated server set comprised of 3f+1 tightly coupled Byzantine-fault-tolerant replicas. Suppose the request cannot be resolved locally and the replicas need to query the root name server or another set of replicated name servers for the answer. When the answer finally reaches replica set 1, the result is cached in the replicas. Now depending on when the query from the original client reached the replica, when the answer arrived at the replica, and when the answer was finally returned to the client, it is possible that answers from the cache of different replicas in the same replica set will have different TTL values, even if all the nodes are non-faulty. It therefore becomes a concern how to establish a certificate [8] of 2f+1 *matching* answers even though some of the answers may have *different* TTL values.

We can think of two different schemes to solve the problem: a server scheme and a client scheme.

Server Scheme

In this scheme, after receiving a query from the client, the designated primary of a view proposes a current time together with the query to all the other replicas. The proposed time is used as the origin time for the specific query. Once the replicas accept the origin time proposed by the primary, using the three-phase protocol described in section 4.3, they resolve the query as usual and retrieve the answer. Now the time to evict the resource record in concern from the cache will be calculated using the origin time proposed by the primary and established at every replica. Since a single origin time is used to calculate the remaining time-to-live (TTL) in all replicas, the answers from different replicas will now have the same TTL value. If the primary is faulty, it will be detected and a different replica will be selected as the new primary using the view-change mechanism described in section 4.3.1.

The advantage of this scheme is that the client does not have to do anything special to establish an agreement among answers with different TTL values. Moreover, the states across the replicas will be the same. The disadvantage, however, is that for every query, the primary needs to propose an origin time and there needs to be an agreement across replicas about the establishment of this origin time. This scheme increases the size of a query because of the inclusion of time in the request packet. For a read-write request, the primary could piggy-back the current time in the *pre-prepare* message it sends to the replicas. However, for a read-only request, which is of most concern to DNS, the client normally multicasts the request to all the replicas and the replicas independently execute the request and send the replies back to the client. In this case the server scheme described above requires an extra phase of communication between the primary and the other replicas to establish the origin time for the query. This adds to the latency of the query resolution process, and increases the overall network traffic because of the additional exchange of messages.

Client Scheme

In the second solution, which we prefer, the replicas resolve the query as usual. While authenticating the replies, they take the TTL field out of the digest and send the final answer to the client. The client collects 2f+1 answers with the same digest, sorts the answers according to the TTL values, and accept the answer with the median TTL value as the final answer.

Since the TTL field is no longer inside the digest, differences in TTL values are not reflected as a difference in answers. On the other hand, the advantage of accepting the answer with the median TTL value as the final answer is twofold:

- the system does not need to make any clock time synchrony assumption across replicas, although the use of leases requires clock rate synchrony assumption.
- in the worst case, the TTL value is > some good answers AND <= some good answers, as explained below.

Let the following be the list of the 2f+1 answers collected by the client after sorting them according to their TTL values:

$$ans_1,\ ans_2,\ ans_3,\ \ldots,\ ans_f,\ ans_{accepted},\ ans_{f+2},\ ans_{f+3},\ \ldots,\ ans_{2f+1}$$

Since there can be at most f faulty answers at one time, there must be at least f+1 good answers out of the 2f+1 answers collected by the client. Under this assumption, we can consider three possible cases:

- **Case I :** The first f answers are faulty. In this case, $ans_{accepted}$ is non-faulty and therefore, the final result is correct.
- **Case II :** The last f answers are faulty. For this case also, $ans_{accepted}$ is non-faulty and therefore, the final result is correct.
- **Case III :** Even in the case where $ans_{accepted}$ is faulty, there are at least f+1 good answers. For this case, TTL value in $ans_{accepted}$ > TTL values in some good answers and <= TTL values in some other good answers, which happens to be a good value for us.

In addition to simplicity, this scheme avoids unnecessary increase in response latency and network traffic because of the additional exchange of messages to establish the origin time across replicas, as required by the server scheme.

However, for the associated checkpoint mechanism to work, this scheme needs a logical definition of system state instead of a physical one so that differences in TTL values across replicas does not get reflected as a difference in states across replicas. In the CLBFT replication library, system state is defined to be a snapshot of the physical memory used by the application. This physical definition of system state has the disadvantage of forcing the exact same contents for a memory subset in every replica, which may be difficult to achieve in the presence non-determinism, even if this non-determinism does not change the state logically. Also, this precludes exploiting design diversity across replicas, since different designs would lead to different physical representations of the state. This makes the replicated system vulnerable to deterministic software errors, which are the most common types of errors in modern systems. A logical notion of state is therefore needed to support any kind of non-determinism in a replicated system. An extended version of the CLBFT algorithm has been implemented by R. Rodrigues [31] of the Laboratory for Computer Science to allow replicas in a Byzantine-fault-tolerant distributed object-oriented database define their state logically to tolerate non-determinism that arises due to certain scheduling of threads. A similar solution can be incorporated to support non-determinism in the Byzantine-fault-tolerant domain name system.

6.4.3 Lease Mechanism

To minimize the window of freshness attack, the system supports a lease-based invalidation scheme, as described in section 5.4.2, to propagate data update notices fast from the source to the sink. The lease is for a certain amount of time, within which the server guarantees to notify the caching client of the data invalidations.

We maintain leases per client. The client uses a *getLease* request to obtain a new lease from the server. The *getLease* request is called when a client has no objects cached from this server, e.g., at the first reference to an object from this server. When a client already has a lease; it uses the *extendLease* request to ask for the lease to be extended.

Both the *getLease* and the *extendLease* requests are *write* requests. Since we need an agreement about what the lease is, the *getLease* and the *extendLease* requests from the client are sent to the server-primary, which chooses the lease period and forwards it to the replicas. Using the three-phase protocol described in section 4.3, the replicas establish an agreement on the lease period. The agreed-upon lease period is then sent to the client. In this way, both the client and server group agree on the lease period.

Care is needed to ensure that the lease is interpreted properly at both the server and the client. Replicas at the server track the lease period using their local clocks; they record their local clock value when the request first arrives, and the lease period expires at a replica when that much time has elapsed at that replica, according to its local clock.

Also replicas at the client note their local time when they make the *getLease* or *extendLease* request. When a client replica receives the response, it computes its own version of the lease period by subtracting the elapsed time since it sent the request from the lease period it receives in the response. A replica at the client knows that the lease has expired when the rest of the lease period is gone according to its local clock.

This way of computing leases only requires that clock rates are synchronized. Under this assumption a lease will expire at a client sooner than it expires at the server because the event of requesting the lease happens earlier than the event of the request arriving at the server.

6.4.4 Invalidation Request

In SBFTDNS, an invalidation request is handled as a typical read-write request in the CLBFT algorithm that uses the three-phase protocol, as described in section 4.3, to atomically multicast the invalidation request to the replicas.

When an authorized server executes an update for some data item it controls, it sends invalidation requests to all replica sets that have active leases. In making the request, the roles are reversed: the authorized server becomes the client and the replica set with the lease is the server.

The authorized server makes an invalidation request by sending a message to the primary of the client holding the lease. A timestamp is used to ensure exactly-once semantics for the execution of the invalidation request. The primary atomically multicasts the invalidation request to all the backups and triggers the three-phase protocol. After executing the invalidation operation, replicas send an acknowledgement of the invalidation to the requester.

After processing an invalidation for some resource record in its cache, the replica group propagates the invalidation to other replica groups that hold leases with it. This cascade of invalidations ensures that notice of updates of security critical data is propagated from the source to the sink as fast as possible to minimize the window of vulnerability.

6.4.5 Dynamic Updates

Data updates are very rare in DNS. The crucial difference between a read request and an update request is that replicas must check the authenticity and the authority of the incoming update request. We use the following authorization policy:

- An update request signed by the private key belonging to a domain name has the authority to update any RRs owned by this domain name.
- An update request signed by a special *zone master key* has the authority to update any RRs in that zone.

In SBFTDNS, dynamic updates are performed using a scheme similar to the lease mechanism described in section 6.4.3. Like lease requests, an update request is handled as a typical read-write request in the CLBFT algorithm.

6.4.6 Reducing the Number of Messages Across Levels of Hierarchy

A potential problem with hierarchically organizing Byzantine-fault-tolerant name servers is the geometric increase in the number of messages with levels of hierarchy as discussed in section 5.3.1. We modified the original CLBFT algorithm to provide a scalable solution to this problem by minimizing the number of messages across levels of hierarchy. The crux of the solution is that when replicas in some replica set need to act as clients to another set of replicas, instead of all the replicas performing the query the primary performs the query on their behalf. This is illustrated in figure 6-5. Depending on the type of request, we have implemented a read-only and a read-write scheme, where the difference originates from the need of authentication.

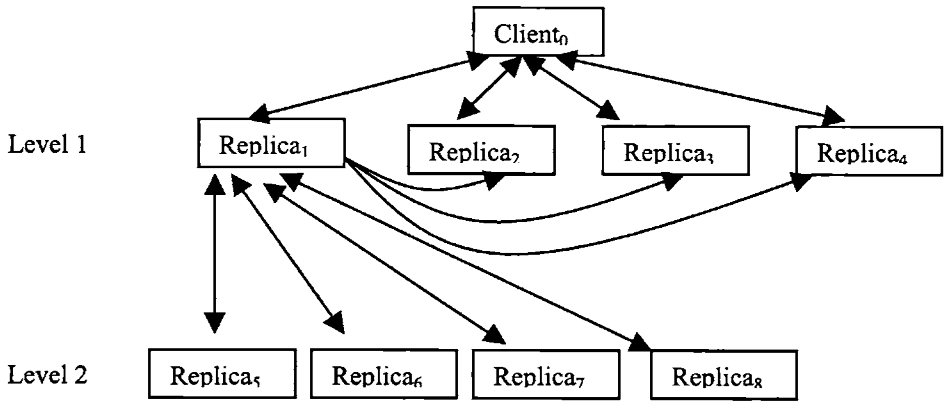


Figure 6-5: A Scheme to Support Replicated Clients with Reduced Number of Messages

While describing the method of authentication, we use the following convention:

K_{in} :	The session key for incoming messages from this replica
K_{out}:	The session key for outgoing messages to this replica

MAC_{in} : Message authentication code generated by using K_{in} as the key
MAC_{out} : Message authentication code generated by using K_{out} as the key

Read-Only Request

A read-only request does not modify the state of the system. Therefore, it is not necessary to verify that a read-only request came from at least f+1 replicas acting as clients. When the original client sends out a read-only request to the 1st level of servers and the request cannot be served at this level, the primary in level 1 authenticates the request with a special flag notifying the replicas in a level 2 server that this is a request on behalf of a set of replicas. The primary then sends the request out to the 2nd level of servers. Replicas in the 2nd level verifiy the request using the key of the primary. In the ordinary case, after resolving the request, each replica authenticates the reply using the key of a single client. However, in this special case, where the client (the primary of level 1) is acting on behalf of all the replicas in level 1, the replicas in level 2 authenticates the reply using the keys for each of the replica in level 1. For example, an authenticated reply from $replica_5$, as illustrated in figure 6-5, will look like:

Reply	$MACout_{1,5}$	$MACout_{2,5}$	$MACout_{3,5}$	$MACout_{4,5}$

The primary for level 1 collects 2f+1 correct replies from the 2nd level. It then generates a reply by appending the MACouts collected from the 2f+1 replies and sends this specially constructed reply to the backups. The backup replicas can verify the authenticity of the reply using the MACs relevant to it. In this way, the system guarantees correctness of behavior to check against a malicious primary. After verifying the reply, the replicas in level 1 send the reply back to the original client. The case of primary being faulty is taken care of by the view change mechanism using the same technique described in section 4.3.1

Read-Write Request

When the replicated clients issue a read-write request to modify the state of the system, it is important to verify that the request came from at least f+1 replicas. Suppose an authorized client in figure 6-6 originates a read-write request to the 1st level of servers. If the replicas in level 1 need to send the request to the 2nd level, we use the designated primary method as described before. However, in the read-only case, the primary needs no explicit designation from other replicas, since level 2 replicas do not care whether the read-only request originated from f+1 1st level replicas or not. In the read-write case, this is crucial. Therefore, after receiving a read-write request, the backup replicas in level 1 explicitly designate the primary by sending it their MAC_{in}s corresponding to the servers of level 2.

Using a similar method to that described above, the primary constructs a special request by appending the MAC_{in}s for f+1 replicated clients in level 1 and sends the request to the 2nd level replicas. After receiving the special request, replicas in 2nd level verify that the request came from f+1 level 1 replicated clients and then execute the request. The rest of the operation is similar to the read-only operation described before.

6.5 Related Work

Our system is based on a similar work done by Zheng Yang [40] at the Laboratory of Computer Science at MIT. To provide Byzantine fault tolerance, Zheng's system comprised of 3f+1 tightly coupled replicas per name server in each zone. The system provides correct domain name service in the presence of up to f faulty replicas.

Our implementation substantially extends Zheng's system. First, we implement a hierarchical organization of name server proxies, rather than a flat organization, to take advantage of a hybrid of iterative and recursive resolution schemes that minimizes the number of session keys. With this scheme, we reduce the cost of expensive operations to establish session keys between the client and the replicas, which would otherwise be too expensive for most practical systems, as discussed in section 5.3.

Second, our system supports the caching of resource records to enhance the performance of DNS. Also, we provide support for replicated clients, an invalidation protocol, and a scheme to reduce the number of messages across hierarchy, which are new additions to the system.

In addition, unlike the old version of the CLBFT library used in Zheng's system, the new version we used in our implementation supports proactive recovery of replicas and the view change algorithm, which affects the *liveness* property of the system [9]. The new version of the replication library also supports packet retransmission mechanism, which lets a client time out and re-send the request in case it cannot collect enough authenticated replies. Since communication in DNS is typically carried by UDP packets for which packet delivery is not guaranteed, some request or reply packets may get lost. In that case, the client may not be able to collect sufficient matching replies to finish a query and need to retransmit the request.

Chapter 7

Performance Evaluation

In this chapter, we provide a performance evaluation of the scalable Byzantine-fault-tolerant domain name system (SBFTDNS). The implemented system supports resource record caching, invalidations, and updates. It also provides support for replicated clients with a scheme to reduce the number of messages across levels of hierarchy, as described in section 6.4.6. The lease mechanism described in section 5.4.2 has not yet been implemented; instead the system uses leases equal to the TTL.

To evaluate the performance of SBFTDNS, we first describe the nature of the experiments and the experimental setup we used to measure the performance of the system. For features that are common, we compare the performance of SBFTDNS with TIS/DNS, and TIS/DNSSEC. We also evaluate the performance of SBFTDNS for additional features it supports, e.g., resource record caching, invalidation and update operations. We measure the performance of the system with and without the support for reducing the number of messages across levels and the session key caching mechanism. Finally, we discuss additional sources of performance optimizations that are likely to be achieved in real systems but are not covered by the experiments we perform.

7.1 Experimental Setup

We evaluated the performance of SBFTDNS under normal case behavior without checkpoint management, view changes, key refreshments, or recovery, because this indicates the most commonly observed behavior of the system.

The experiments ran on nine Dell Precision 410 workstations with a single Pentium III processor, and 512 MB of memory. All machines ran Linux 2.2.16-3. The processor clock speed was 600MHz in seven machines ad 700 MHz in the other two. The machines were connected by a 100 Mb/s switched Ethernet and had 3Com 3C905B interface cards.

We use the Rabin-Williams public-key cryptosystem with a 1024 bit modulus to establish 128-bit session keys. All messages are then authenticated using message authentication codes (MAC) computed using these keys and UMAC32 [4]. Message digests are computed using MD5 cryptographic hash function [29].

7.2 Experiments

Throughout the experiments, we compared the performance of SBFTDNS with TIS/DNSSEC, an implementation of the DNS security extension proposed by the IETF and implemented by the Trusted Information System. TIS/DNSSEC is based on the BIND version 4.9.4-pl, the most popular DNS package. We added the reply verification module to TIS/DNSSEC as the original implementation lacked the crucial support of verifying the authenticity of the reply by the client. To compare the performance of our system with the non-secure DNS, we compiled TIS/DNSSEC with the security mode off, and call it "TIS/DNS". The test systems are denoted as follows:

System	Means of Authentication
TIS/DNS	No authentication
TIS/DNSSEC	Pre-generated SIG RRs signed by the zone public key
SBFTDNS	MACs generated by session key that is established by using the server private-public key pair, with additional support for key refreshment, replicated client, RR caching, and invalidation

In the absence of a standard benchmark to measure the performance of the domain name system, we devised our own experiments. For all the experiments, the zone databases were loaded into memory and we used hash tables to store the database records. Therefore, the core database lookup procedure took very little times regardless of the size of the database. The experimental zone database contained about 120 resource records in it.

Our experiments comprised of two major types of operations:

7.2.1 Read Only Operations

Read only operations that do not modify the state of the system are the most commonly requested operations in the domain name system. We measure the performance of the two most common types of read-only requests, queries for A records and NS records.

A (Address) Record Query

We request the IP address of *host1.foo.bar* from the name servers of the zone *foo.bar*. Depending on the system, an A record and an optional SIG RR are returned to the client. This is the simplest possible query in DNS, and is also the most common type of request. As indicated by the data in appendix A, almost 60.16% of DNS queries going out of MIT's Laboratory for Computer Science in January, 2000 were A record queries.

NS (Name Server) Record Query

We request NS records for the name servers of the zone *foo.bar* to the servers of the zone *bar*. In TIS/DNSSEC, there are two name servers for each zone, whereas in SBFTDNS, there are four name servers per zone. Generally, this type of query returns a lot of resource records, including NS RRs, A RRs corresponding to the name servers, related KEY RRs, and optional SIG RRs. Although quite complicated, this type of query is also quite common since the resolvers need to know the name servers of a zone to query the zone.

7.2.2 Read-Write Requests

Although quite infrequent in current DNS, read-write operations like dynamic updates are getting popular due to the use of DNS to support server load distribution and host mobility. We perform two types of read-write requests: resource record invalidation and update operations. We perform the simplest update operation of changing the IP address of a host in the zone foo.bar. For invalidation, we performed a single level and a multilevel invalidation of a cached resource record. Since TIS/DNS and TIS/DNSSEC do not support RR update or invalidation schemes, we only measure the performance of SBFTDNS for these two types of operations.

7.3 Results

In this section, we describe the results we obtained from running 10000 invocations of each operation. For these experiments, we assume that the session keys between the client and the replicas in SBFTDNS are established in the very first invocation, and they are cached for the rest of 9999 operations. This amortizes the cost of session key establishment and the obtained results indicate the performance of the system when the session key is cached. Later in section 7.4.1 we discuss the performance of SBFTDNS when the session key is not cached.

7.3.1 Read-only Operations

Query for an A Record

Table 7-1 summarizes the performance of the test systems for a query for the IP address of hostt1.foo.bar. The *#RR* signifies the total number of resource records included in the reply. *#SIG* indicates the number of SIG RR (included in *#RR*) included in the response packet to authenticate the reply. The *elapsed time* is the response latency observed by the client from the time the client starts to generate a request to the time it accepts a reply and successfully interprets it. The *server time* is measured from the time the server receives a request to the time the reply packet is ready to be sent out, i.e., it does not include the time it takes to receive a request or send a reply.

System	#RR / #SIG	Reply Size (bytes)	Elapsed Time (ms)	Server Time (ms)
TIS/DNS	1/0	47	0.146	0.029
TIS/DNSSEC	2/1	150	0.359	0.035
SBFTDNS	1/0	112	0.253	0.036

Table 7-1: Performance for the Query for an A Record

In the non-secure TIS/DNS, the response for an A record includes only one A RR and no SIG RR to prove the authenticity of the reply. The answer in TIS/DNSSEC includes one A RR and its corresponding SIG RR. SBFTDNS, on the other hand, authenticates a reply using message authentication codes (MAC) instead of SIG RR. The response therefore includes only one A RR and no SIG RR.

The results show that response latency for TIS/DNS is lower than TIS/DNSSEC. The difference arises mostly from the cost of verifying a SIG RR at the client. We measured that each SIG RR verification operation (dnssec_verify) in TIS/DNSSEC costs 0.018 ms. Also, the reply size has almost tripled from TIS/DNS (47 bytes) to TIS/DNSSEC (150 bytes). Processing a larger response packet increases latency both at the client and the server.

As expected, the client elapsed time and the server processing time in SBFTDNS are larger than those of TIS/DNS. The extra cost is introduced by the cryptographic operations to verify the authenticity of the reply and the overhead of the replication library that involves extra computation and the processing of larger request/reply packets. However, the A record reply verification cost at the BFT client is found to be 0.007 ms (including 0.003 ms of generating digest and 0.004ms for verifying MAC), which is significantly lower than 0.018 ms of the SIG RR verification cost in TIS/DNSSEC. The reply size is also smaller, i.e., 112 bytes compare to 150 bytes. This results in a smaller elapsed time (0.25 ms vs 0.35ms) in SBFTDNS compared to that in TIS/DNSSEC.

Query for NS Record

Table 7-2 summarizes the performance of a query for the name servers of the zone *foo.bar*.

System	#RR / #SIG	Reply Size (bytes)	Elapsed Time (ms)	Server Time (ms)
TIS/DNS	4/0	98	0.19	0.043
TIS/DNSSEC	9/4	592	1.04	0.072
SBFTDNS	12/0	552	0.44	0.087

Table 7-2: Performance for the Query for an NS Record

For both TIS/DNS and TIS/DNSSEC, we assume two names servers per zone. Therefore, the response from TIS/DNS includes two NS RRs and two A RRs corresponding to the IP addresses of the name servers. Because of a bigger answer set, the latencies are larger than what we found for A record queries.

In case of TIS/DNSSEC, the response includes two NS RRs, one SIG RR for the NS RRs, two A RRs, two SIG RRs for the A RRs, one KEY RR for the zone and one SIG KEY RR. Because of the larger answer set with four SIG RRs, which involves costly verification operations, the client elapsed time has substantially increased from 0.36 ms to 1.04ms.

In case of SBFTDNS, we assume four replicas acting as the server set for a zone. The response includes four NS RRs, four A RRs, and four KEY RRs corresponding to each replica. The number of RRs is larger than an A record reply. However, the response latency has not increased substantially (0.44ms compared to 0.25ms). The difference between the performance of TIS/DNSSEC and SBFTDNS is quite noticeable in this case. This is due to the fact that TIS/DNSSEC needs to verify four SIG RRs using expensive public-key cryptography. With each additional RR, the cost increases substantially. The client in SBFTDNS, on the other hand, needs to verify only one MAC for each reply, regardless of the number of RRs in the reply message. The MACs are computed using session keys and UMAC32, the cost of which is significantly less than that of a public key cryptographic operation. Given the popularity of NS record queries, our system can provide a service with significantly higher performance, availability, and security than TIS/DNSSEC or similar DNS security extension implementation using public key cryptography.

7.3.2 Read-Write Operations

Resource Record Update

Table 7-3 shows the performance of SBFTDNS for an RR update operation. Since TIS/DNS or TIS/DNSSEC do not support update transactions, we only tested the performance of SBFTDNS.

System	Elapsed Time (ms)	Primary Server Time (ms)	Non-primary Server Time(ms)
SBFTDNS	0.5168	0.299	0.133

Table 7-3: Performance for Updating the IP address of a Host

For this experiment, we assumed that the session key for dynamic update had been cached. Later in section 7.4.1 we discuss the performance of an update operation when the dynamic update key has not been cached.

The results show an increase in both the client elapsed time and the server processing time, compared to the read-only operations. However, as we discussed in section 5.4, data updates are very rare in DNS. The crucial difference between a read request and an update request is that replicas must check the authenticity and the authority of the incoming update request. Notice the distinction between the performance of the primary of a view and the non-primary replicas of SBFTDNS system. A read-write operation like RR update is performed as a typical transaction in the CLBFT algorithm. When the primary receives a request, it uses a three-phase protocol (*pre-prepare, prepare* and *commit*) to atomically multicast the request to the backups, as described in section 4.3. Because of the request authentication, the extra message roundtrip and extra cryptographic operations performed by the replication library, the server time increases substantially in both the primary and the non-primary replicas. The overall response latency observed by the client, however, is not too high when the session key is in cache.

Resource Record Invalidation

Table 7-4 shows the performance of SBFTDNS for the invalidation of a resource record that has been cached at the servers. We measured the performance for both a one-level and a two-level invalidation. In the one-level case, an authorized client makes an invalidation request to replica set 1 for the ns record of some.bar that has been cached in the replicas as a result of a previous query. However, replica set 1 did not provide the cached answer to any other server set; therefore invalidation stops with one level. The two-level case measures the performance of the system when replica set 1 receives an invalidation request from an authorized client for a cached resource record. After, verifying the authenticity of the client, replica set 1 invalidates the resource record from its own cache and sends an authenticated invalidation request to replica set 2 which cached the RR from replica set 1 and whose lease period has not expired in the mean time. For the two-level case, we show the performance of the system for with and without the scheme for reducing the number of messages across levels.

Mode	Elapsed Time (ms)	Primary Server Time (ms)	Non-primary Server Time(ms)
One-Level	0.506	0.287	0.128

Table 7-4: Performance for One-Level Invalidation Operation

Like any other read-write request, invalidation is performed as a typical transaction with a three-phase protocol, which explains the increase in the server processing time. The results indicate that the performance for a single level invalidation is almost identical to the update request (cost is slightly higher for update), as both the cache and the database are loaded in memory. Manipulating resource records stored in hash tables take a very small amount of time.

Mode	# Message Reduction Mode	Client Elapsed Time (ms)	Replica Level 1		Replica Level 2	
			Primary Server Time (ms)	Non Primary Server Time (ms)	Primary Server Time (ms)	Non Primary Server Time (ms)
Two-Level	Off	1.64	1.057	1.053	0.3	0.13
	On	1.70	1.346	1.371	0.355	0.134

Table 7-5: Performance for Two-Level Invalidation Operation

The two-level case is interesting. The measured client elapsed time here indicates the amount of time it takes for the invalidations to be committed both at replica set 1 and 2. The latencies at both the server and the clients increase due to additional round trip time between levels and extra cryptographic operations at the server side. The data also show that invalidation latencies increase when the message reduction mode is on due to extra operations needed to indicate the specialty of the request, increase in the size of the replies, and additional communication between the primary and the non-primaries to delegate the responsibility. Later in section 7.4.3, we discuss in detail the effect of reducing the number of messages across levels.

7.3.3 System Performance Comparison

Client Elapsed Time

The client elapsed time indicates the response latency observed by the client. This is one of the most important metric of DNS performance evaluation. Figure 7-1 and 7-2 indicate the relative performance of TIS/DNS, TIS/DNSSEC, and SBFTDNS in terms of response latency for an A record and an NS record query.

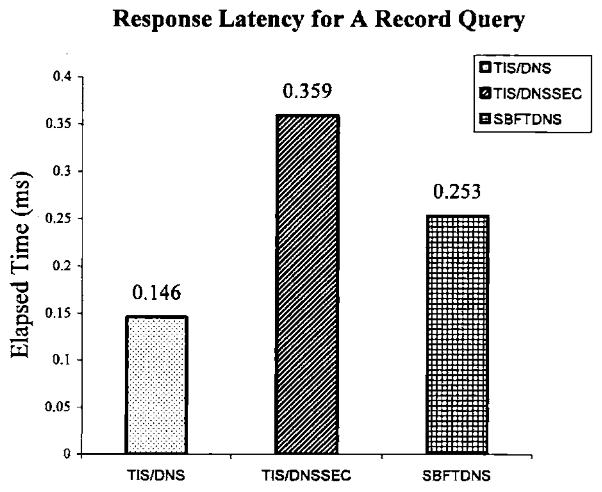


Figure 7-1: Response Latency for A Record

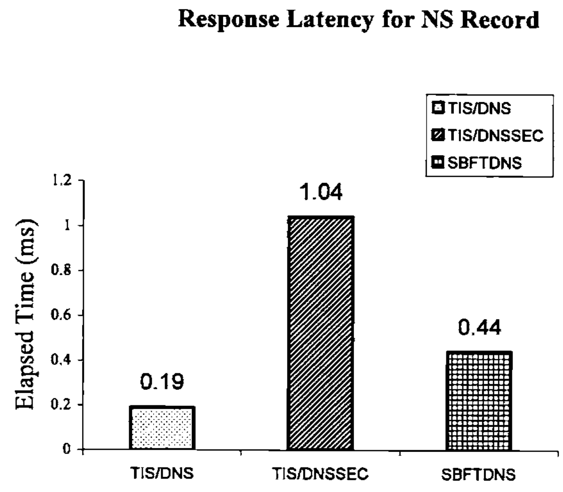


Figure 7-2: Response Latency for NS Record

For A Record query, the elapsed time in SBFTDNS is 4% higher than TIS/DNS. However, it is 4.2% lower than the elapsed time in TIS/DNSSEC. The gain is even higher for a sufficiently large NS record query that involves a lot of cryptographic operations. Our test results show a 136% gain in response latency for NS record query while using SBFTDNS instead of TIS/DNSSEC. The performance here is only 5.7% lower than the insecure version of TIS/DNS.

7.4 Performance Issues

7.4.1 Session Key Management

We now discuss the performance of the system when the session key is not cached in the client and the replicas. As discussed in section 5, we use the Rabin-Williams public-key cryptosystem with a 1024-bit modulus to establish a 128-bit session key. Table 7-7 shows the amount of time it takes to encrypt, decrypt, sign and verify a session key using the Rabin-Williams public-key cryptosystem.

Operation	Time (ms)
Encrypt	0.91
Decrypt	28.4
Sign	29.3
Verify	0.12

Table 7-7: Cost of Cryptographic Operations in the CLBFT

Table 7-8 shows the performance of SBFTDNS with and without session key caching.

Operation	Session Key in Cache	Client Elapsed Time (ms)	Server Time (ms)
A Record Query	YES	0.253	0.036
A Record Query	NO	32.4	28.5
A Record Update	YES	0.517	0.299
A Record Update	NO	62.5	29.01

Table 7-8: Performance of SBFTDNS with and without Session Key in Cache

The results indicate that the both the client elapsed time and the server time increase significantly when the session key is not in cache. For a read-only request like an A record query, client authentication is not important. In this case, the client performs four encryption operations and each replica performs a decryption, which is 30 times slower than an encryption. Therefore, the server becomes the bottleneck.

For a read-write request like an A record update, however, the authentication and authorization of the client is important. The client needs to provide a signature to prove its authenticity and authority, in addition to the 4 encryptions operations. Therefore, the total elapsed time is almost double the elapsed time in the read-only operation.

In fact, the data show that performance of the system when session key is not cached is at least 95% slower than when session key is cached. But even then, the amount of time it takes at the client and the server to perform an invalidation/update operation is not high when we compare it to the usual cost of network latency. Given the infrequency of update/invalidation requests, the performance is acceptable.

7.4.2 Replicated Clients and RR Caching

Because of DNS's use of hierarchical state partitioning and caching to achieve scalability, we had to develop an efficient protocol that allows the replicas in a group to request resource records from another group of replicas and cache the result for future reference. Table 7-9 shows the effect of RR caching on the performance of SBFTDNS. In the first case, the client requested the NS record of some.bar, which could not be provided by the replica set 1. The first level of replicas then obtained the answer from replica set 2 (the authoritative name servers for the zone bar), cached the result and sent it back to the original client. For 10000 invocations of the request, replica set 1 had to obtain the record from the 2nd level only once, and it served the answer from its own cache for the rest of 9999 requests. This essentially amortizes the cost for obtaining the result from the second level, and indicates the performance of the system when it serves an RR from its local cache.

RR Caching	Client Elapsed time (ms)	Replica set 1 Server Time (ms)	Replica Set 2 Server Time (ms)
ON	0.28	0.047	0.04
OFF	0.9	0.6	0.038

Table 7-9: Performance of SBFTDNS with RR caching on and off

In the second case, we turned the caching mode off. Therefore, the first level of replicas had to obtain the answer from the second level each time. Due to the extra round trip and processing time at the 2nd level servers, the response latency experienced by the client increases from 0.28 ms to 0.9 ms, almost a 221% increase. This emphasizes the need for RR caching for overall performance benefit of a scalable domain name system.

7.4.3 Reducing the Number of Messages across Levels of Hierarchy

The results in table 7-5 indicate that the client elapsed time increased from 1.64ms to 1.7ms (a 3.6% increase) when the scheme for reducing the number of messages across levels was turned on. The increase can be attributed to the extra operations needed to indicate the specialty of the request, increase in the size of the replies, and additional communication between the primary and the non-primaries to delegate the responsibility.

The gain, however, comes from the reduction of server load, network overhead and traffic, especially with the increase of the level of hierarchy. Table 7-10 shows the number of messages and an approximate number of bytes traversing through the network for a read-only request for a one, two and three level hierarchical system, with and without the reduction scheme. We assume that each server level comprises four replicas to tolerate one failure. The RR caching mode is assumed to be off, which may be the case for a security critical data. We consider a standard network header size of 20 bytes. The reply size, S is assumed to be 160 bytes, which is the average DNS response packet size we observed in the LCS traffic, as described in appendix A. The MAC size is 10 bytes and the reply representation takes 40 bytes (besides the actual reply data) in the CLBFT replication library.

Reduction Scheme	**# of Levels**	**MaximumTotal # of Messages traversing through the Network**	**Total # of reply bytes traversing the Network (Reply size, S=160 bytes)**
ON	1	4*2 = 8	(20+S+40+10)*4 = 920
	2	4*2 + 4*2 + 3 = 19	(20+S+40+40)*7 + (20+S+40+10)*4 = 2740
	3	4*2 + 4*2 + 4*2 + 3*2 = 30	(20+S+40+40)*7*2 + (20+S+40+10)*4 = 4560
OFF	1	4*2 = 8	(20+S+40+10)*4 = 920
	2	4^2*2 + 4*2 = 40	(20+S+40+10)*20 = 4600
	3	4^3*2 + 4^2*2 + 4*2 = 168	(20+S+40+10)*84 = 19320

Table 7-10: Number of Messages and Reply bytes for a Read-only Request

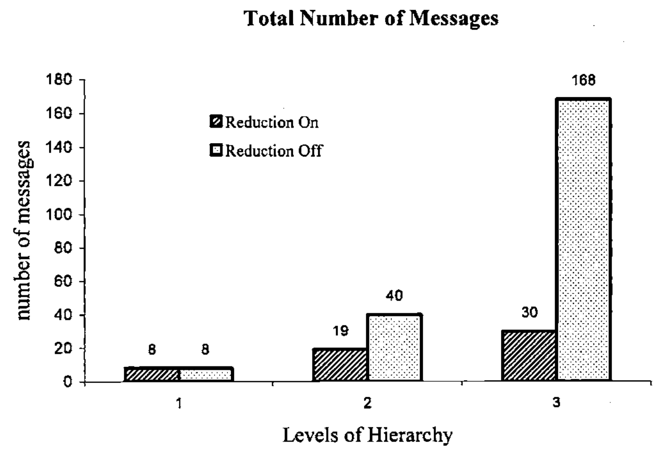


Figure 7-3: Total number of Messages

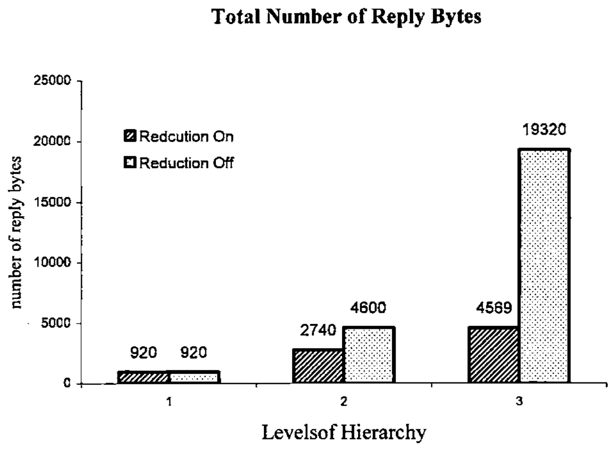


Figure 7-4: Total number of Reply Bytes

Even with a two-level hierarchy, the reduction scheme almost halves (19 instead of 40) the number of messages, and yields a 67% reduction in the total number of reply bytes (2740 bytes instead of 4600 bytes) traversing through the network.

For a three-level system, the gain is even higher, as the number of messages is cut by a sixth (30 instead of 168), and the total number of reply bytes has been reduced to one fourth of what it would be without the reduction scheme (4560 bytes instead of 19320 bytes).

In addition, the scheme significantly reduces the load on the servers as we ensure that each replica in level n receives only one request from the replicas in level n-1. This reduces the load on the 2nd level replicas by a factor of 4 (1 request instead of 4). The load on the 3rd level replicas is cut down by a factor of 16. We consider this to be a significant gain that improves the scalability of a replicated system as ours.

7.5 Discussion

In this chapter, we presented a comparative performance evaluation of the scalable Byzantine-fault-tolerant domain name system. It should be noted that the lease requests proposed in section 5.4.2 have not been implemented in the system. However, this does not affect the performance results we obtained. Our experiments were run with one client and two levels of servers, each level comprising four replicas to tolerate one fault. It took less than 5 seconds to serve 10000 invocations of each query. Even if we assume that a typical lease is for one minute, the experiments did not last long enough for leases to be extended. For the 10000 invocations of a read request, we would have had one *getLease* request and no *extendLease* requests. Compared to the total number of queries, the cost for a *getLease* request would not make any difference.

Our test results show that SBFTDNS performs worse that the insecure DNS, but substantially better than TIS/DNSSEC in almost every case, not to mention that SBFTDNS guarantees a much higher level of security, robustness, and availability of the domain name system than TIS/DNSSEC. The performance gain increases with the number of cryptographic operations, due to the elimination of expensive public-key cryptography and the use of session-key based message authentication codes (MAC) in our system.

We observed that the performance of the system when the session is cached is significantly better than when the session key is not cached. However, even the performance of the system when the session key is not cached seems not too high when we take typical network latency into account. As a further optimization, we use the Rabin-Williams public-key cryptosystem to establish the session key, which is at least 5 times faster than the RSA Public-key cryptosystem used in TIS/DNSSEC for cryptographic operations [9].

In reality, we can expect our system to perform even better due to various other optimizations in the CLBFT algorithm. For example, our experiments were designed to invoke 10000 operations from the same client. In reality, server throughput will be much higher due to the request batching support in the CLBFT library. With batching, the replication algorithm can process many requests in parallel. This amortizes the protocol overhead for read-write operations over many requests. The server throughput with batching grows with the number of clients up to a maximum that is 66% better than the throughput with a single client [9]. The algorithm also uses optimizations to reduce protocol overhead as the operation argument and return sizes increase.

Chapter 8

Conclusions

In this thesis, we presented the design, implementation, and performance evaluation of a scalable Byzantine-fault-tolerant domain name system. In section 8.1, we summarize our work and major contributions. Section 8.2 proposes research directions for future work.

8.1 Summary

The domain name system is the standard mechanism on the Internet to advertise and access important information about hosts, e.g., address, mail exchanger, etc. The original DNS was not designed to be a secure protocol. Because of its critical role in the Internet infrastructure, DNS has become a favorite target of malicious attacks in recent times. Due to the lack of support for data integrity authentication and source authentication in the current DNS protocol, a resolver has no alternative but to trust the result it receives. Furthermore, current DNS does not tolerate server failures due to benign faults or malicious attacks.

With the growing popularity of the Internet, and the reliance of various Internet protocols on DNS data, it is important that DNS tolerate failures and attacks both at the servers and during data communication over the Internet. In late 1997, a security extension of the domain name system (DNSSEC) was proposed by the Internet Engineering task force (IETF). The basic idea is to provide data integrity and origin authentication to security aware resolvers by means of cryptographic digital signatures.

While the proposed DNS security extension (DNSSEC) ensures a secure communication of DNS data, DNSSEC assumes that the zone private key is not stolen and the servers that are responsible for providing authenticated name service are not corrupted themselves. We discussed the security holes in DNSSEC infrastructure in detail in chapter 3. The biggest concern in DNSSEC is where to store the zone private key, which is used to sign the zone data. If the zone key is stored online, a single name server failure may subvert the entire zone. If stored offline, however, dynamically updated data is either not available or not protected before the next signing takes place, which defeats the purpose of dynamic update. Furthermore, the wide window of freshness attack due to the use of pre-generated signature resource records (SIG RR), vulnerability to both benign and malicious server failures, and the use of a simple primary-secondary replica scheme in zone transfer causing inconsistency between primary and secondary name servers, add to the disadvantages of the proposed DNS security extension.

As a solution to these problems, we propose a scalable Byzantine-fault-tolerant secure domain name system. By using the CLBFT replication library, we provide high availability and integrity of the domain name system in the presence of Byzantine faults, i.e., arbitrary behavior exhibited by the nodes. Each name server in our Byzantine-fault-tolerant DNS consists of 3f+1 tightly coupled replicas to survive up to f faults within a small window of vulnerability. In this way, we remove the vulnerability of the system to both benign and malicious server failure, increase the reliability and availability of the domain name service, and ensure consistency of state among non-faulty replicas.

By using a session key mechanism, we authenticate communication between a client and a server, and provide per-query data authentication. Since no pre-generated signature is used, data provided by the name servers is always up-to-date. In addition, SBFTDNS supports real-time dynamic data updates, and dynamically updated data are as safe as any other data in SBFTDNS.

The major concern with any session key based system, however, is scalability. Since every client and server needs to share a session key to protect communication between them, the number of session keys increases quadratically with the number of nodes. The cost of establishing session keys using expensive PPK operations can be reduced by incorporating an efficient session key caching mechanism. Based on current patterns of DNS traffic, we propose a hierarchical organization of name servers to take advantage of a hybrid recursive and iterative query resolution approach. The scheme minimizes the number of session keys to remember, as well as reducing the load on the root name servers.

On the other hand, DNS, being a very large-scale distributed system, makes extensive use of resource record caching. Because most operations in DNS are read-only and data updates are very infrequent, RR caching is a favorite technique to achieve scalability and performance gain. Caching, however, raises concern about cache consistency and staleness of data since stale data may not be harmless when security is of concern. We reduce the window of freshness attack by implementing an efficient hierarchical caching scheme with an invalidation protocol using leases.

A Byzantine-fault-tolerant implementation of the domain name system is particularly interesting because of DNS's use of hierarchical state partitioning and caching to achieve scalability. To implement a Byzantine-fault-tolerant DNS, we had to develop an efficient protocol for replicated clients that allows replicas in a group to request operations from another group of replicas. To provide a scalable solution to the problem of increase in the number of messages across levels of replicated clients, we developed a protocol where the primary performs the query on behalf of the group. To remove vulnerability to a faulty primary, the protocol ensures that the reply from one set of replicas can be verified by another set independently of who performed the query.

Finally, in chapter 7, we presented an evaluation of the performance of SBFTDNS. We compared the performance of our system with non-secure DNS, and with TIS/DNSSEC, an implementation of the domain name security extension protocol. While SBFTDNS performs worse than the insecure DNS due to extra cryptographic operations to support security, the results show that SBFTDNS performs as well or better than TIS/DNSSEC in almost every case, not to mention that SBFTDNS guarantees a much higher level of security, robustness, and availability of the domain name system than TIS/DNSSEC. We observed a 4.2% gain in response latency for a typical A record query, and a 136% gain for an NS record query while using SBFTDNS instead of TIS/DNSSEC. The performance gain increases with the number of cryptographic operations, due to the elimination of expensive public-key cryptography and use of session-key based message authentication codes (MAC) in our system. As discussed in section 7.5, we can expect our system to perform even better in reality due to various other optimizations in the CLBFT algorithm, e.g., request batching and reduction in protocol overhead as the operation argument and return sizes increase.

The proposed scalable Byzantine-fault-tolerant domain name system provides a highly secure service, yet it is practical. Together with high reliability and availability, SBFTDNS has the potential to not only provide a secure name service for the Internet, but also serve as a robust infrastructure for storing authenticated public keys for other internetworking protocols that require authentication.

8.2 Future Work

8.2.1 Improving the Implementation

There is much scope for improvements and optimizations in the scalable Byzantine-fault-tolerant domain name system that we implemented. For many features, like the support for replicated clients, we provide a simple implementation to show the feasibility, and we measured the performance of the system under such a scheme. Also, the lease mechanism described in section 5.4.2 has not yet been implemented.

An important point to note is that in DNS, servers are generally replicated to ensure redundancy and availability of service, and to spread the load among the replicas. For a heavily-used zone, there are many servers and they are needed to support the load. This is an important way of increasing the scalability of servers. However, in a Byzantine-fault-tolerant DNS, a server comprises 3f+1 replicas, and all the replicas execute queries. Although our performance results show that our four-replica server can compete with one DNSSEC server, an obvious concern is how SBFTDNS can handle the load in a heavily-used zone. Further research is needed into how to improve the capabilities of the replicas in SBFTDNS so that they can handle higher loads.

8.2.2 Use of a Standard DNSSEC Package

We based the implementation of SBFTDNS on TIS/DNSSEC beta version 1.4, because at the time of this implementation, it was the only implementation of DNSSEC available. The Internet Software Consortium recently released BIND version 9.0 and declared TIS/DNSSEC to be obsolete [20]. The very latest release, BIND 9.0.1, is capable of acting as an authoritative server for DNSSEC secured zones. This functionality is believed to be stable and complete except for lacking the support for wildcard records in secure zones. When acting as a caching server, BIND 9.0.1 can be configured to perform DNSSEC secure resolution on behalf of its clients.

Since BIND version 9.0 is a more complete implementation of the DNSSEC protocol compared to TIS/DNSSEC, it would be interesting to implement our Byzantine-fault-tolerant domain name system on top of BIND 9.0 and compare the performance of the two systems. Measuring performance against BIND would give a more realistic picture. Any improvement in security and performance over BIND can be viewed as significant since BIND is estimated to be the DNS software used by over 90% of the hosts in the Internet.

8.2.3 Backward Compatibility

One important step in a realistic implementation of a Byzantine-fault-tolerant domain name system is to make the system backward compatible to already existing DNS implementations. The request and reply message formats for a CLBFT replication library are different from the formats in current DNS. Moreover, the BFT client requires 2f+1 matching replies from different replicas to get a convincing answer in SBFTDNS.

To make the system backward compatible, we can use a scheme similar to the one proposed by DNSSEC. According to DNSSEC protocol, a security aware server sets the AD (authenticated data) bit in the response header to indicate that the answer has been authenticated by a security aware server. A security aware resolver, on the other hand, may trust its local name server, in which case it would issue a query that looks identical to a non-security-aware resolver's query. It would, though, expect the AD bit set in the response. Transaction security is needed in this case to prevent malicious tampering over the local link. A security aware resolver may also issue a query with a CD (checking disabled) bit set, which would instruct the local name server to forgo any authentication checks. The resolver would then perform the security checks itself.

In the Byzantine-fault-tolerant case, a BFT server should be able to recognize requests from old servers correctly and must send back recognizable replies. A BFT resolver, on the other hand, must be able to know whether the name server it wants to contact is Byzantine fault tolerant or not, and should prepare the request message accordingly. A possible solution would be to incorporate some additional information in the zone NS record to indicate the type of the zone.

Also, in our implementation the hierarchical organization of replicas was configured manually using configuration files. It would be useful to provide a more generic solution to the problem of organizing name servers in hierarchy by extending the *forwarder* directive provided by domain name system software like BIND.

8.2.4 Realistic Performance Measurements

We have measured the performance of the system in the normal case. It would be interesting to see how the system behaves when faults are injected into the system. Measuring the performance and behavior of the scalable Byzantine-fault-tolerant DNS under view change, failure recovery, and packet retransmission mode would be useful. Also, more comprehensive experiments, e.g., a multi-client set up instead of a single client one, can better approximate the behavior of the system in reality.

Appendix A

An Analysis of DNS Traffic Patterns and Performance

In this section, we present an analysis [1] of the current DNS traffic patterns, performance, and caching behavior, as observed from network traffic traces. The objective is to study the typical distribution of DNS request latencies, identify potential bottlenecks in the name resolution schemes, and investigate how effective DNS caching behavior is. The motivation is to understand the current behavior of DNS to propose ways to improve the over performance of the domain name system.

A.1 Motivation

Because of the critical role that the domain name system plays on the Internet, virtually every internetworking service use DNS, and share their fate with it. It is therefore important to improve the overall performance of DNS by minimizing the request latency and by eliminating possible bottlenecks in typical DNS query resolution schemes.

The traditional approach to reduce request latency in DNS is caching resource records. There are apparently contradictory claims about the expected behavior of an efficient DNS caching scheme. On one hand, aggressive caching is considered to be beneficial, since caching reduces the load on the name servers and decreases the latency observed by the resolvers. To make full use of this feature of caching, the TTL values of DNS resource records should not be short compared to the rate of change of actual binding. On the other hand, the use of DNS as a level of indirection to balance load among several servers, or newly emerging services like host mobility calls for very short resource record TTLs. In a recent work [33] done by Emil Sit of MIT's Lab for Computer Science (LCS), the caching behavior of DNS has been studied. It investigates the impact of sharing caches and varying TTLs on DNS cache hit rates. The paper suggests that sharing a DNS cache among many clients is not very effective at increasing hit rate. Similarly, while TTLs of a less than a few seconds substantially harm hit rates, TTLs beyond a few minutes have little impact on the hit rate. Although this seems to have a negative implication on the effectiveness of caching, it suggests that the use of DNS in a dynamic and non-cached mode need not greatly degrade its performance. In our analysis, we further extend some of the questions addressed by this work [1] and focus on various per-query based statistics, e.g., request latency and number of referrals involved in a typical query. We also study several characteristics of DNS behavior in general, including query types, popularity and response errors. To figure out the impact of sharing on cache hit rates, we perform a trace driven simulation.

In a related work in 1992, Danzig et al.[11] presented measurements of DNS traffic at a root name server. They found that one third of the wide-area DNS traffic that traversed the NSFnet in 1992 was destined to seven globally distributed root name servers. Moreover, they estimated that DNS consumed approximately twenty times more wide-area network bandwidth than was absolutely necessary. This was more than ten times the amount of excess traffic estimated from the observations of DNS traffic in 1988 [24]. This motivates us to investigate whether root name servers are the bottlenecks of current DNS name resolution schemes. We analyze what percentage of outgoing DNS queries contact a root name server and how those queries are distributed among thirteen root servers. We also investigate the effect of contacting root name servers on DNS performance.

A.2 DNS Traces

A.2.1 Data Collection

Our analysis is based on traffic traces collected on the border router of the MIT Laboratory for Computer Science (LCS) on a sole link that connects the LCS with the rest of MIT's network. The traces are taken from January 3rd,2000 to January 9th, 2000 and will be referred to as the *week1* trace for the rest of this paper. The detailed description about data collection can be found in [33].

A.2.2 Analysis Methodology

DNS traces are analyzed to extract various statistics including the number of referrals involved in a typical query and the distribution of request latency for various types of query resolution patterns. To calculate the latency in resolving a query, we maintain a sliding window of 8 seconds to match a list of queries sent out of the LCS with incoming response packets. If a response contains an answer record, we look for a query that matches with that response and output the time difference between those two packet arrivals. The actual end-user DNS request latency, however, is longer than this, since we see packets in the middle of the resolvers and the name servers. If an answer record is absent from a response, we increment the corresponding query's number of referrals by one and wait until the final answer comes. To keep track of the name servers contacted during a query resolution, each query has an array storing the IP addresses of name servers involved in the query resolution.

A.3 Analysis

A.3.1 General Characteristics

The basic statistics about DNS traffic is summarized in table A-1. Our analysis is based on the LCS *week-1* traces. More details about the general trace attributes including query rate, average packet size and number of queries per one outgoing TCP connection are described in [33].

	All	Result :No Error	Result:Error
Total Queries	1770865	1489432(84.1%)	281433(15.9%)
% of queries to mit.edu	11.5%		
% of queries to root	18.99%	18.56%	21.27%
% of answers from root	28.77%		
% of referrals from root	71.23%		
Traffic Statistics	Value		
Average Total packets/sec	≈9000		
Average TCP connection/sec	10.14		
Average DNS packets/sec	25.94		
Average DNS query size (bytes)	39.9		
Average DNS response size(bytes)	160.8		

Table A-1: Basic Statistics

As shown in table A-1, we found that almost 19% of the queries end up querying a root name server. Moreover, 28.77% of the times a root server gets contacted, it provides the final answer to the query, which seems unusual given that root servers are not supposed to perform recursive queries. These two facts taken together indicate that the load on the root name servers is quite high.

A.3.2 Request Latency

Figures A-1 and A-2 show DNS request latency distribution. About 10% of the requests get their responses in less than 10ms and the median value is 75ms, which means DNS performs quite well in most cases. When we exclude queries to the local domain, *mit.edu*, the peak at 1ms of the solid line of figure A-1 disappears. That peak is specific to the location of the data collection machine of the *week1* traces. We observed about 3% of requests that take more than 1 second. The peak around 8 second in the figure A-1 corresponds to our window size in the analysis because we don't keep un-responded query packets longer than 8 seconds.

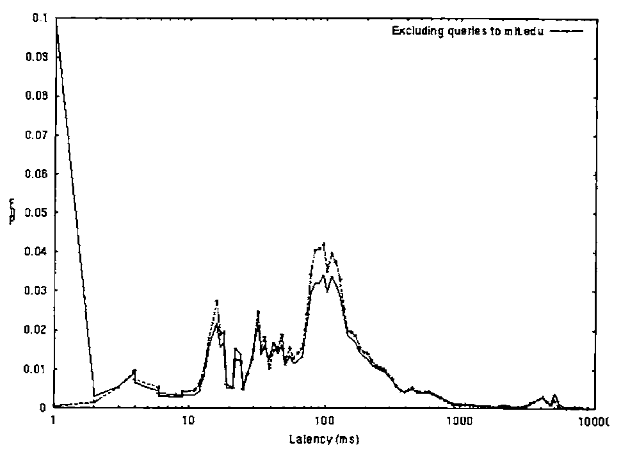


Figure A-1: PDF of DNS request latency

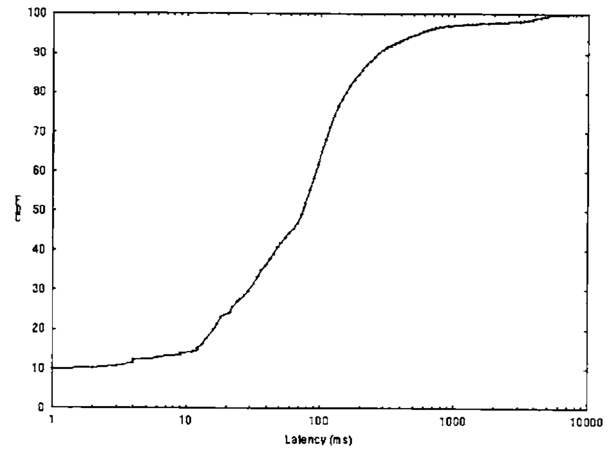


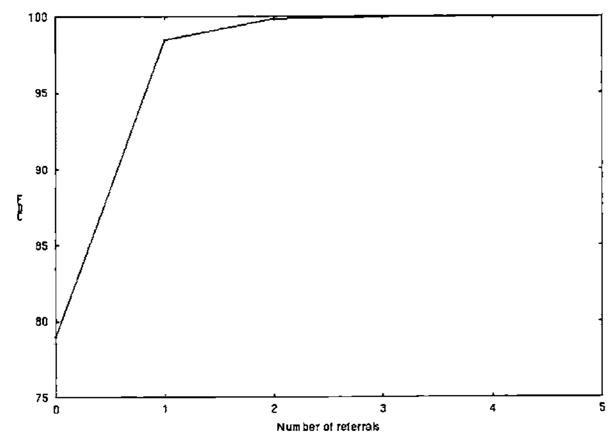
Figure A-2: CDF of DNS request latency

A.3.3 Number of Referrals

Referrals happen when a server does not know the answer to a question, but does know where the answer can be found. In that case, it sends a response packet with the information in it. DNS request latency, therefore, highly depends on the number of referral packets since the resolver repeatedly contacts a series of name servers to get an answer. Figure A-3 shows the distribution of referral packets for a query. 79% of queries are resolved without any referral, which means they get an answer directly from a server it contacts the first time. There are 0.01% of them, which take more than 5 referrals, but these cases can be considered pathological due to errors, such as DNS server or resolver misconfigurations.

0 referral	78.99%
1 referral	19.48%
2 referrals	1.40%

Table A-2: Number of referrals



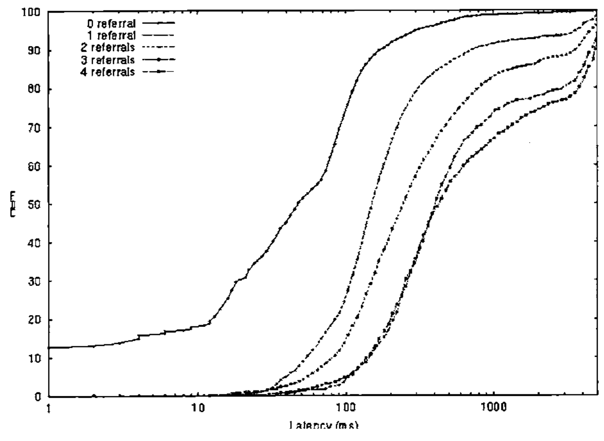


Figure A-3: CDF of the number of referrals

Figure A-4: Latency distribution vs # of referrals

Figure A-4 shows the latency distribution according to the number of referrals. It is obvious that latency goes higher as the number of referrals increases. For the case of 0 referral, 73% of requests are resolved in less than a 100ms and only 1% of requests take more than 1 second. However, more than 84% of requests take more than a 100ms if they get more than 2 referrals in resolving the query. Therefore, reducing the number of referrals involved in the resolution of a query, can improve DNS request latency significantly.

A.3.4 Query Type

A query type is specified in the question section of DNS packets. Table A-3 lists top four most frequently requested query types found in the *week1* traces. Most of the observed queries were seeking the IP address corresponding to a host name (60.160% A query) or vise versa (34.16% PTR query).

We find that about 30% of PTR queries are made to the name servers in the mit.edu domain, which may be the result of inverse look up of IP-addresses, which happens, for example, when ssh connections are established from a machine in the mit.edu domain to any host inside the lcs.mit.edu network. Figure A-5 shows the cumulative distribution of request latencies across query types.

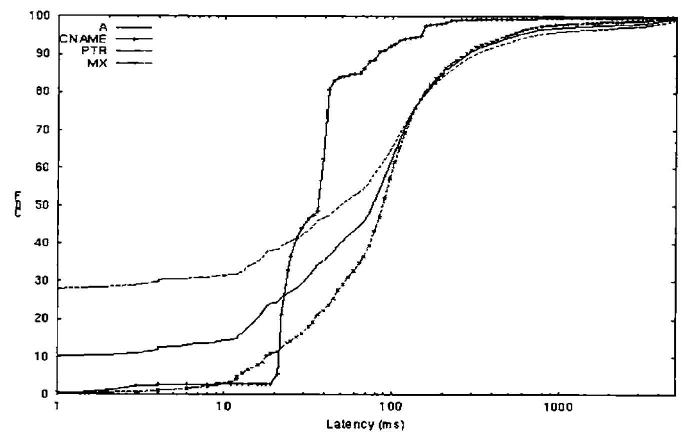


Figure A-5: Latency distribution across query types

Type	Frequency
A	60.16%
PTR	34.16%
CANAME	2.52%
MX	3.11%
Other	0.05%

Table A-3: Query types observed

A.3.5 Response Error

As shown in table A-1, 15.9% of the queries in the *week1* trace result in an error as the response. According to [26], the response return code, RCODE is set as a part of the response indicating the types of errors occurred at the name server while resolving a query. Table A-4 shows that most of the errors are composed of NXDOMAIN or SERVFAIL errors. The NXDOMAIN signifies that the domain name referenced by the query does not exist, while the SERVFAIL indicates that the name server is unable to process the query due to a problem with the server. Inspecting the error cases further, we find that non-negligible cases of errors are caused by inappropriate domain name expansion (e.g: *dirty.research.bell-labs.com.lcs.mit.edu*) and wrong inputs such as *loopback*, *cow*, or *15.0.29.18.rbl.maps.vix.com*. Moreover, some error cases are highly repetitive which can be avoided by negative caching, thereby improving the response time and offloading the name servers at the same time.

NXDOMAIN	73.70%
SERVFAIL	25.61%
Other	0.69%

Table A-4: Response errors observed in *week1* trace

A.3.6 Root Server Characteristics

There are 13 well-known root servers, which are contacted when a local proxy name server cannot resolve a query using its local database or cache. Table A-5 lists the root servers and the number of queries forwarded to them, as observed in the *week1* trace.

Root Server	Location	Access Count
A	Herndon, VA, USA	11255 (3.35%)
B	Marina Del Rey, CA, USA	16858(5.01%)
C	Herdon, VA, USA	20231(6.01%)
D	College Park, MD, USA	166199(49.4%)
E	Mt View, CA, USA	21805(6.48%)
F	Palo Alto, CA, USA	19133(5.68%)
G	Vienna, VA, USA	8829(2.62%)
H	Aberdeen, MD, USA	24715(7.34%)
I	Stockholm, Sweden	15725(4.69%)
J	Herndon, VA, USA	2219(0.65%)
K	London, UK	1655(0.49%)
L	Marina Del Rey, CA, USA	997(0.29%)
M	Tokyo, Japan	952(0.28%)

Table A-5: Root name servers

Figure A-6 shows the latency distribution across those 13 root name servers. It shows that each root server has different performance characteristics and load is not evenly distributed across those name servers, at least for the outgoing LCS DNS traffic.We clearly see that D.ROOT-SERVERS.NET has the best performance, so more than half of the requests are forwarded to that server corresponding to the server selection rule described in RFC 1035.

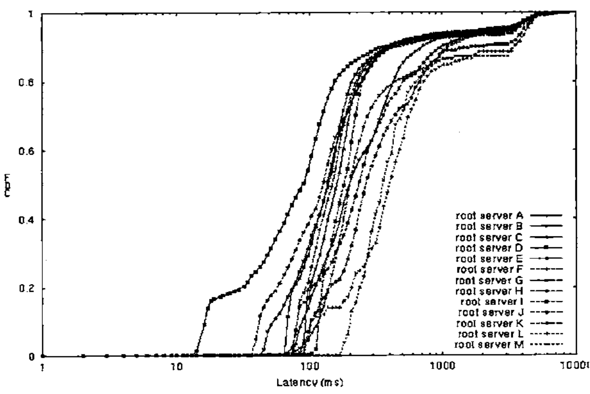
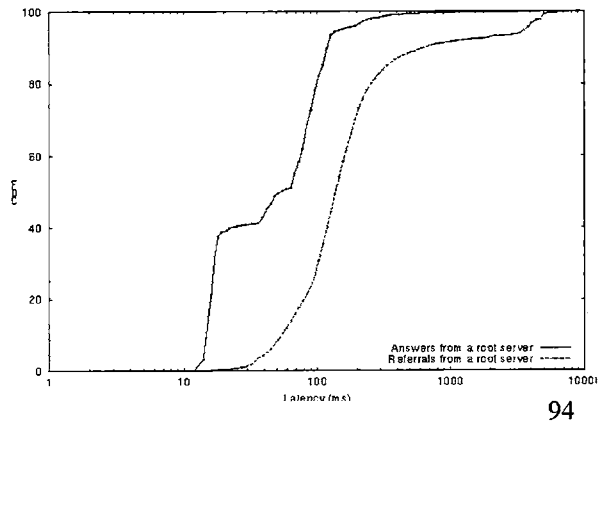


Figure A-6: Latency distribution across roots

In figure A-7, we compare the latency distribution of queries for which the root name server provides the final answer, and queries that get referrals from the root name servers. It takes at least 10ms to resolve a query that ends up querying a root name server. Performance goes even worse when referrals from the root servers are involved. This motivates DNS to use caching extensively to avoid contacting



root servers. In the next section, we examine how caching is beneficial in terms of reducing the request latency.

Figure A-7: Latency when root servers return answers vs. referrals

A.3.7 Caching NS Records

To understand the effect of caching, DNS queries are classified as either a *hit* or a *miss* by lookingup the first server contacted by the resolution of the query. We assume a *miss* when a query is directly forwarded to one of the root servers. Otherwise, we assume that there is a *hit* for
an NS records in DNS cache, which bypasses querying a root name server. Figure A-8 shows the latency distribution for each case. It clearly shows that caching NS records substantially reduces DNS request latency even though it may involve some referrals to complete the query resolution.

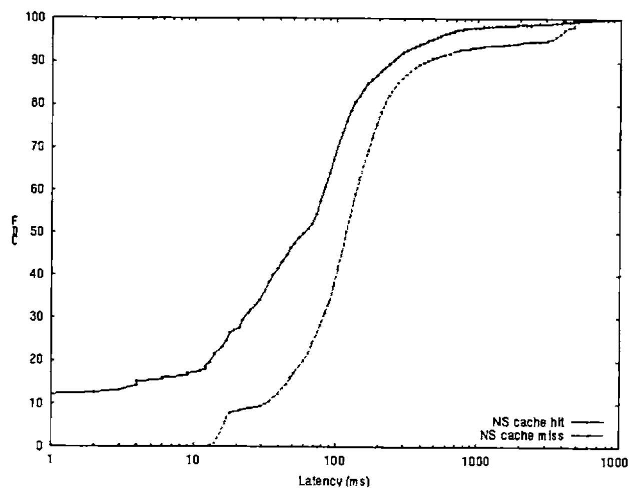


Figure A-8: Cache Hit and Miss

A.3.8 Effect of Aggregation on Cache Hit Rate

To determine the impact of sharing on cache hit rate and how much it gains over independent per-client caching, we did a trace driven simulation of cache behavior under different aggregation conditions. To make the simulation more realistic, we collected DNS responses from the trace to form a database of IP addresses and associated TTL values as included in the response. Also, to handle cases where a domain name is associated with multiple IP addresses, e.g., for load balancing purposes, we generated a database containing the destination IP addresses of TCP connections and the domain names they map to.

We randomly divide 962 TCP clients into groups of size s. Each group is assigned a cache containing IP address/TTL mappings and DNS names to be shared by all members of the group. If a client c is making a TCP connection to destination IP address i, we search if i is already present in the IP address cache of the group of which c is a member. If i is found in the cache and the associated TTL has not expired, then it is considered a hit. If i is found in the cache, but the TTL has expired, then it is considered a miss. We then refresh the entry for i and its associated domain name in the cache with the TTL of i found from the database. If i is not found in the cache, we then search the database to find the domain name that IP address i maps to. If the name is found in the name cache with a valid TTL, we consider it a cache hit and add i to the IP address cache with the remaining TTL of the name entry in the name cache. This takes advantage of the fact that multiple IP addresses associated with the same domain name are included in the same DNS response and therefore cached together. If the name is found in the cache and its TTL has expired, it is a miss. We then add i to the IP cache and set the TTL to the actual TTL value found from the database. In the end, we record the total number of cache hits, total number of requests and classify different types of misses.

Multiple IP-Name	1	2	10	100	All
On	64.0%	76.2%	82.9%	86.9%	91.0%
Off	60.0%	72.2%	77.8%	81.2%	84.9%

Table A-6: Effect of Aggregation on cache hit rate

For each level of aggregation, we ran four independent simulations, once considering references to different IP addresses that map to the same domain name as hits (according to the algorithm described above) and once considering a one-to-one mapping between IP addresses and domain names (each IP address is considered distinct while considering cache hit/miss). Table A-6 summarizes the data obtained from the simulation. In the absence of any sharing, the average per-connection cache hit rate is 64% with multiple-IP-name on, and about 60% with multiple-IP-name off. This indicates that a 4% gain in cache hit rate can be obtained, on a per-connection basis, due to references made by the same client to different IP addresses mapping to the same domain name. As the data suggests, increase in client group size s significantly increases the cache hit rate. When all the 962 clients were simulated to share the same cache, we observe an average hit rate of 91% with multiple-IP-name on, and 85% with multiple-IP-name off.

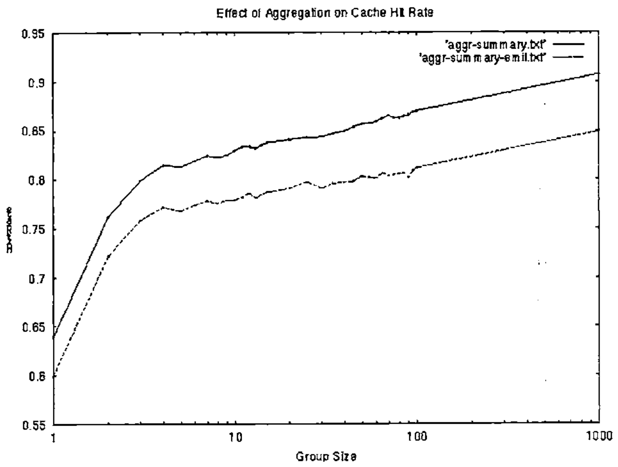


Figure A-9: Effect of aggregation on cache hit rate

Based on these results, we conclude that sharing significantly improves cache hit rates, and thereby DNS performance. By aggregating clients into groups to share the same cache, we can achieve better hit rates than a single client cache. Moreover, when all the clients are aggregated into one group, we observe that 84.9% of all the misses were TTL misses, i.e., misses for names that were previously reached by some client in the same group, whereas the remaining 15.1% were compulsory misses caused by a first time access to an IP address. This indicates that increasing the TTL values holds the potential for increasing cache hit rates, thereby improving the overall DNS performance.

A.4 Discussion

From the collected DNS traces, we observed that the DNS request latency increases significantly with the number of referrals. With the use of extensive caching and improved TTL values, the number of referrals could be minimized, which would increase the overall performance of DNS. Also, the load on the root name servers, although less than what was observed in 1992 [11], is found to be still too high and contacting a root name server significantly increases the request latencies. While investigating how beneficial caching is to reduce the request latencies, we found that caching NS records significantly reduce latency, thereby improving DNS performance. The trace-driven simulation indicated that sharing of caches among clients substantially increases DNS cache hit rates. We observed cache hit rate to increase from 63.9% when using per-client caching to 91% when all the clients were sharing the same cache. Together, these call for a more efficient implementation of the domain name system, which holds the potential for offloading the root name servers by taking advantage of an effective query resolution scheme and a smarter caching mechanism that takes benefits from sharing among clients.

References

[1] S. Ahmed and J. Jung. *An Analysis of DNS Traffic Patterns, Performance, and Caching Behavior.* Final Project, Computer Networks, Massachusetts Institute of Technology, 2000.

[2] R. Alonso and M. Blaze, *Dynamic Hierarchical Caching for Large-scale Distributed File Systems.* The 12th Intl. Conference on Distributed Computing Systems, June 1992

[3] P. Albitz and Cricket Liu. *DNS and BIND*, Third Edition. O'Reilly and Associates, Inc., 1998

[4] J. Black, S. Halevi, H. Krawczyk, T. Krovetz, and P. Rogaway. *UMAC: Fast and Secure Message Authentication.* Advances in Cryptograpy – CRYPTO'99.

[5] C. Mic Bowman, Peter B. Danzig, Darren R. Hardy, Udi Manber, Michael F. Schwartz, and Duane P. Wessels. *Harvest: A Scalable, Customizable Discovery and Access System.* Technical Report CU-CS-732-94,Department of Computer Science, University of Colorado, Boulder, August 1994 (revised March 1995).

[6] M. Castro and B. Liskov. *Practical Byzantine Fault Tolerance.* In Proceedings of the Third Symposium on Operating Systems Design and Implementation (OSDI99), New Orleans, USA, February 1999

[7] M. Castro and B. Liskov. *Authenticated Byzantine Fault Tolerance Without Public-Key Cryptography.* Technical Memo MIT/LCS/TM-589, MIT Laboratory for Computer Science, June 1999

[8] M. Castro and B. Liskov. *Proactive Recovery in a Byzantine Fault Tolerant System.* In Proceedings of the Fourth Symposium on Operating Systems Design and Implementation (OSDI 2000), San Diego, USA, October 2000

[9] M. Castro. *Practical Byzantine Fault Tolerance.* PhD Thesis, Massachusetts Institute of Technology, November 2000.

[10] M. D. Dahlin, R. Y. Wang, T. E. Anderson, D. A. Patterson. *Cooperative Caching: Using Remote Client Memory to Improve File System Performance.* Proc. First Symposium on Operating Systems Design and Implementation. pp. 267-280. November 1994. Also appeared as University of California Technical Report CSD-94-844.

[11] P. Danzig, A. Chunkhunthod, K. Worell, C. Neerdaels, M. Schwartz, *A Hierarchical Internet Object Cache.* Technical Report 95-611, Computer Science Department, University of Southern California, Los Angeles, California, March 1995. Also, Technical Report CU-CS-766-95. Department of Computer Science, University of Colorado, Boulder, Colorado.

[12] DNS Resource Directory available at http://www.dns.net/dnsrd

[13] D. Eastlake. *Secure Domain Name System Dynamic Update.* Technical Report DARPA-Internet RFC 2137, Cybercash Inc., April 1997

[14] D. Eastlake. *Domain Name System Security Extensions.* Technical Report DARPA-Internet RFC 2535, IBM, March 1999

[15] D. Eastlake. *DSA Keys and SIGs in the Domain Name System.* Technical Report DARPA-Internet RFC 2536, IBM, March 1999

[16] D. Eastlake. *RSA/MD5 Keys and SIGs in the Domain Name System.* Technical Report DARPA-Internet RFC 2535, IBM, March 1999

[17] D. Eastlake and C. Kaufman. *Domain Name System Security Extensions.* Technical Report DARPA-Internet RFC 2065, Cybercash Iris. January 1997

[18] C.Gray and D. Cheriton. *Leases: An Efficient Fault-Tolerant Mechanism for Distributed File Cache Consistency.* In Proceedings of the 12th ACM Symposium on Operating System Principles, pages 202--210, December 1989.

[19] Internet Software Consortium, available at http://www.isc.org

[20] Internet Software Consortium BIND (Berkeley Internet Name Domain) 9.0 information, available at http://www.isc.org/products/BIND/bind9.html

[21] M.L. Kazar, *Synchronization and Caching Issues in the Andrew file System.* Usenix Conference Proceedings, Dallas, Winter 1998

[22] L. Lamport, R. Shostak, and M. Pease. *The Byzantine Generals Problem.* In ACM Transactions on Programming Languages and System, 4(3), 1982

[23] *Managing Servers with Netscape Console: Introduction to SSL* available at http://developer.netscape.com/docs/manuals/security/sslin/contents.htm

[24] P. Mockapetris and Kevin J. Dunlap. *Development of the Domain Name System.* 1988

[25] P. Mockapetris. *Domain Names – Concepts and Facilities.* Technical Report DARPA – Internet RFC 1034, ISI, Nov 1987.

[26] P. Mockapetris. *Domain Names – Implementation and Specification.* Technical Report DARPA – Internet RFC 1035, ISI, Nov 1987.

[27] Network Association inc. Secure DNS Product, 1999. Available at http://www.nai.com/products/security/tis_research/netsec/net_dns.asp

[28] Network Solutions Inc, 1999. Available at http://www.networksolutions.com

[29] R. Rivest. *The MD5 Message-Digest Algorithm.* Internet RFC-1312, April 1992.

[30] R. Rivest, A. Shamir, and L. Adleman. *A Method for Obtaining Digital Signatures and Public-Key Cryptosystems.* In Communications of the ACM, V. 21, N. 2., February 1978

[31] R. Rodrigues. Private Correspondence, 2000.

[32] J. Saltzer, D. Reed, and D. Clark. *End-To-End Arguments in System Design.* ACM Transactions on Computer Systems, Vol 2, No. 4, November 1984.

[33] E. Sit. *A Study of Caching in the Internet Domain Name System.* Master's thesis, Massachusetts Institute of Technology, 2000.

[34] F. Sneider. *Implementing Fault Tolerant Service using the State Machine Approach: a Tutorial.* In ACM Computing Surveys, 22(4), December 1990.

[35] A. Snoeren and H. Balakrishnan. *An End-to-End Approach to Host Mobility.* In proceedings of the 6th ACM/IEEE International Conference on Mobile Computing and Networking (MobiComm'00), 2000.

[36] K. Thompson, G. Miller, and R. Wilder. *Wide-area Internet Traffic Patterns and Characteristics.* IEEE Network, 1997.

[37] P. Vixie, S. Thompson, Y. Rekhtar, and J. Bound. *Dynamic Updates in the Domain Name System.* Technical Report DARPA- Internet RFC-2136 ISC, Bellcore, Cisco, DEC, April 1997

[38] B. Wellington. *An Introduction to Domain Name System Security.* TISLabs at Network Associates, January 22, 1999

[39] K. J. Worrell. *Invalidation in Large Scale Network Object Caches*. PhD thesis, University of Colorado-Boulder, 1994.

[40] Z. Yang, *Using a Byzantine Fault Tolerant Algorithm to Provide a Secure DNS*. Masters Thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, June 1999