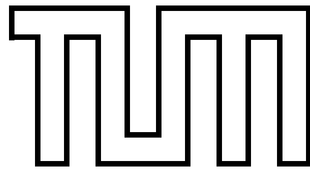


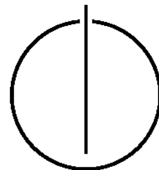
FAKULTÄT FÜR INFORMATIK

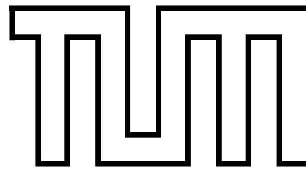
DER TECHNISCHEN UNIVERSITÄT MÜNCHEN

Masterarbeit in Informatik

Design and Implementation of a Censorship Resistant and Fully Decentralized Name System

Martin Schanzenbach, B.Sc.





FAKULTÄT FÜR INFORMATIK

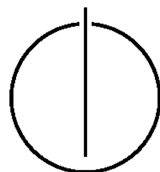
DER TECHNISCHEN UNIVERSITÄT MÜNCHEN

Masterarbeit in Informatik

Design and Implementation of a Censorship Resistant and Fully Decentralized Name System

Design und Implementierung eines zensurresistenten und vollständig dezentralisierten Namenssystems

Author:	Martin Schanzenbach, B.Sc.
Supervisor:	Christian Grothoff, Ph.D. (UCLA)
Advisor:	Dipl.Inform. Matthias Wachs
Date:	September 25, 2012



I assure the single handed composition of this master's thesis only supported by declared resources.

Munich, September 24, 2012 Martin Schanzenbach, B.Sc.

Acknowledgments

This work is based on a paper with the title "A Censorship Resistant and Fully Decentralized Replacement for DNS" from Christian Grothoff, Matthias Wachs and Martin Schanzenbach.

I thank Christian Grothoff and Matthias Wachs for their extensive support and invaluable advice throughout the making of this thesis. I also thank everyone who submitted information about their browser history for the study on surfing behavior. I thank Jacob Appelbaum for suggestions to improve our design for firewall-based DNS interception and Ludovic Courtès, Ralph Holz, Luke Leighton, Simon Josefsson, Nikos Mavrogiannopoulos, Ondrej Mikle, Stefan Monnier, Niels Möller, Chris Palmer, Martin Pool, Richard Stallman, Neal Walfield and Zooko Wilcox-O'Hearn for insightful comments, compiled as FAQ in Appendix A.

Finally, I want to thank my family for making it possible to start and successfully finish my studies in the first place.

Abstract

This thesis presents the design and implementation of the GNU Alternative Domain System (GADS), a decentralized, secure name system providing memorable names for the Internet as an alternative to the Domain Name System (DNS). The system builds on ideas from Rivest's Simple Distributed Security Infrastructure (SDSI) to address a central issue with providing a decentralized mapping of secure identifiers to memorable names: providing a global, secure and memorable mapping is impossible without a trusted authority. SDSI offers an alternative by linking local name spaces; GADS uses the transitivity provided by the SDSI design to build a decentralized and censorship resistant name system without a trusted root based on secure delegation of authority.

Additional details need to be considered in order to enable GADS to integrate smoothly with the World Wide Web. While following links on the Web matches following delegations in GADS, the existing HTTP-based infrastructure makes many assumptions about globally unique names; however, proxies can be used to enable legacy applications to function with GADS.

This work presents the fundamental goals and ideas behind GADS, provides technical details on how GADS has been implemented and discusses deployment issues for using GADS with existing systems. We discuss how GADS and legacy DNS can interoperate during a transition period and what additional security advantages GADS offers over DNS with Security Extensions (DNSSEC). Finally, we present the results of a survey into surfing behavior, which suggests that the manual introduction of new direct links in GADS will be infrequent.

Zusammenfassung

Diese Arbeit präsentiert das Design und die Implementierung des GNU Alternative Domain System (GADS), einem dezentralisierten, sicheren Namenssystem, welches einprägsame Namen für das Internet bietet, als Alternative zu dem Domain Name System (DNS). GADS baut auf Ideen von Rivest's Simple Distributed Security Infrastructure (SDSI) auf, um das zentrale Problem zu lösen, dezentralisiert sichere Bezeichner einprägsamen Namen zuzuordnen: Das Bereitstellen einer globalen, sicheren und einpräsamen Zuordnung ist unmöglich ohne globale, vertrauenswürdige Instanz. SDSI bietet hierfür eine Alternative. GADS nutzt transitive Delegation von lokalen Namensräumen, bereitgestellt durch das SDSI Design, um ein dezentralisiertes, zensurresistentes Namenssystem zu erschaffen, dass keine zentralen Vertrauensanker benötigt. Dieses Ziel wird erreicht durch sichere Delegation von Zuständigkeit.

Zusätzliche Details müssen bei der Integrierung von GADS in das World Wide Web beachtet werden. Während das Folgen von Links im Web den Delegierungen in GADS entspricht, macht die existierende HTTP-basierte Infrastruktur viele Annahmen in Bezug auf global einzigartige Namen; durch den Einsatz von Proxies ist es jedoch möglich, dass Applikationen, die solche Annahmen machen, mit GADS funktionieren.

Diese Arbeit stellt die fundamentalen Ziele und Ideen hinter GADS vor, liefert technische Details zur Implementierung von GADS und erläutert Probleme bei dem Einsatz von GADS in existierenden Systemen. Es wird dabei erörtert, wie die Interoperabilität von GADS und DNS in einer Übergangsperiode gewährleistet werden kann und welche zusätzlichen Sicherheitsvorteile GADS gegenüber DNS mit seinen Sicherheitserweiterungen (DNSSEC) bietet. Schliesslich werden die Ergebnisse einer Umfrage zum Surfverhalten von Nutzern diskutiert, welche nahelegt, dass das manuelle Einführen neuer direkter Verbindungen in GADS nur gelegentlich nötig ist.

Contents

Acknowledgments **vii**

Abstract **ix**

1. Introduction **1**
1.1. Contribution . . . 1
1.2. Adversary Model . . . 2
1.3. Organization . . . 2

2. Related Work **5**
2.1. Simple Distributed Security Infrastructure . . . 5
2.2. Name Systems . . . 6
2.2.1. Petname Systems . . . 8
2.2.2. Domain Name System . . . 9
2.2.3. DNSSEC . . . 11
2.2.4. Tor .onion System . . . 12
2.2.5. Timeline Systems . . . 14
2.3. Distributed Hash Tables . . . 15
2.3.1. Whanau . . . 15
2.3.2. X-Vine . . . 16
2.3.3. R^5N . . . 18

3. The GNU Alternative Domain System **21**
3.1. Design of the Name System . . . 21
3.1.1. Zone Delegation . . . 22
3.1.2. Network Protocol and Routing . . . 23
3.1.3. Signatures, Expiration and Freshness . . . 23
3.1.4. Three Zones for Security, Privacy and Usability . . . 25
3.1.5. Globally Unique and Secure Names . . . 26
3.1.6. Zone Revocation . . . 27
3.1.7. Context dependent Names . . . 28
3.1.8. Names and Record Types . . . 28
3.2. Integration with Legacy Applications . . . 33
3.2.1. Surfing with Pseudonyms and Petnames . . . 33
3.2.2. Virtual Hosting and SSL Certificates . . . 34
3.2.3. Enabling Replies (e-mail) . . . 35
3.2.4. Accessing GADS without Installation . . . 36
3.2.5. Incompatible Applications . . . 37

4. Implementation **41**
4.1. Integration into Operating Systems . . . 41
4.1.1. Firewall-based DNS Interception . . . 41
4.1.2. DNS-to-GADS Gateway . . . 42
4.1.3. NSS Plugin . . . 43
4.2. GNUnet Name System . . . 44
4.2.1. The Namestore Implementation . . . 44
4.2.2. Network Integration . . . 45
4.2.3. The VPN Service . . . 48
4.3. Complementary Tools and Programs . . . 49
4.3.1. Command-Line Tools . . . 49
4.3.2. HTTP Proxy . . . 51
4.3.3. The GADS Zone Editor and GADS QR codes . . . 55
4.4. Integration into Applications . . . 56
4.4.1. DNS packets . . . 56
4.4.2. GNS API . . . 57
4.4.3. Fork-and-exec . . . 57

5. Discussion **59**
5.1. Establishing Trust with GADS . . . 59
5.2. Automated Name Shortening and Security . . . 60
5.3. Usability and Bootstrapping . . . 61
5.4. Improved Migration for Legacy Networks . . . 61
5.5. Usability Evaluation: Surfing Behavior . . . 62
5.6. Alternative GADS resolvers . . . 64

6. Conclusion and Future Work **67**

Appendix **71**

A. Frequently Asked Questions **71**

B. Command-Line Tool Reference **81**
B.1. gnunet-namestore (1) . . . 81
B.1.1. Name . . . 81
B.1.2. Synopsis . . . 81
B.1.3. Description . . . 81
B.1.4. Options . . . 81
B.1.5. Bugs . . . 82
B.1.6. See Also . . . 82
B.2. gnunet-gns (1) . . . 82
B.2.1. Name . . . 82
B.2.2. Synopsis . . . 82
B.2.3. Description . . . 82
B.2.4. Options . . . 82

B.2.5. Bugs 83
B.2.6. See Also 83

C. GNUnet Name System API **85**
C.1. Function Documentation 85
C.1.1. GNUNET_GNS_cancel_get_auth_request 85
C.1.2. GNUNET_GNS_cancel_lookup_request 85
C.1.3. GNUNET_GNS_cancel_shorten_request 85
C.1.4. GNUNET_GNS_connect 85
C.1.5. GNUNET_GNS_disconnect 86
C.1.6. GNUNET_GNS_get_authority 86
C.1.7. GNUNET_GNS_lookup 86
C.1.8. GNUNET_GNS_lookup_zone 87
C.1.9. GNUNET_GNS_shorten 87
C.1.10. GNUNET_GNS_shorten_zone 88

D. GADS Record Types and Flags **89**
D.1. Record Types 89
D.2. Record Flags 90

E. Browsing Survey **91**
E.1. Scripts 91
E.1.1. Chromium and Chrome 91
E.1.2. Firefox 92
E.2. User Data 94

Bibliography **97**

1. Introduction

"The Domain Name System is the Achilles heel of the Web.
The important thing is that it's managed responsibly. "
– Tim Berners-Lee

The Domain Name System (DNS) is a unique distributed database and a key service for most Internet applications. While DNS is distributed, it relies on centralized, trusted registrars to provide globally unique names. The use of globally unique names causes significant frustration for new users that find that many reasonable names are already owned. As the awareness of the central role DNS plays on the Internet rises, various institutions are engaged in legal attacks on the DNS which threaten the availability and integrity of the Internet [14].

One current effort to secure DNS is DNSSEC [2]. DNSSEC is designed to provide data integrity and origin authentication for replies to DNS queries. The adversary model of DNSSEC excludes legal attacks. Governments, corporations and their lobbies can legally compel operators of DNS authorities to manipulate entries and certify the changes. Soghoian and Stamm warned that this might happen for X.509 certificates [49]. As a result of its adversary model, DNSSEC fails to prevent issues such as the recent brief disappearance of thousands of legitimate domains due to a data management accident in the execution of established censorship procedures [24].

We advocate a solution to those issues in line with the ideas of GNU. Richard Stallman, founder of the GNU project, writes [50]: "When a program has an owner, the users lose freedom to control part of their own lives." In contrast to ownership-based name systems, this paper presents the design and implementation of the GNU Alternative Domain System (GADS), a replacement for DNS which is fully decentralized and eliminates ownership of names, thereby eliminating most[1] legal attacks on the name system and offering users relief from DNS censorship.

1.1. Contribution

Our contribution is a petname system [51] where every individual user is able to freely and securely map names to values. As a result, mappings cannot be globally unique. Our approach borrows a central idea from Rivest's Simple Distributed Security Infrastructure (SDSI) [46]. SDSI is a distributed public-key infrastructure without a global name space where every participant is a *certification authority* issuing his own certificates. Certificates

[1] We say "most", because prosecuting individual users for using software remains a possible legal attack vector.

are used for delegation; they certify the mapping of local names to public keys of other participants.

A central idea in GADS is the replacement of the DNS root zone and DNS delegation with individual petnames assigned by each user in combination with SDSI-like secure delegation of subdomains. The ability to delegate is central to achieving our goal of providing a practical alternative to globally unique names. In addition, GADS uses a censorship-resistant Distributed Hash Table as a decentralized database. The contribution of this thesis is thus a fully decentralized name system which provides the following features:

- *Usability*: cryptographic keys identify entities and are bound to memorable names
- *Security*: malicious participants cannot prevent the creation of such bindings or censor existing bindings
- *Transitivity*: names can be passed conveniently between users, following common network-usage patterns; specifically, if Alice has a name for Bob, and Bob has a name for Carol, then Alice will also have a name for Carol.
- *Compatibility*: the system is compatible with the existing DNS and provides all key capabilities of DNS (except globally unique names)

1.2. Adversary Model

Our adversary model is significantly different from that of DNSSEC as the adversary may participate in any possible role in the system and there is no bound on the percentage of malicious participants. This definition excludes the possibility of having a trusted third party. The adversary can attempt to take over control of names using legal means, for example by confiscating names or forcing operators to direct people to adversary-controlled impostor sites.

Our adversary model allows the adversary to assume multiple identities (Sybils) and to have more computational resources than benign users, including all benign users combined. However, the adversary is unable to break cryptographic primitives and cannot prevent communication between benign participants.

1.3. Organization

The remainder of this work is structured as follows. In Chapter 2, we systematically position related designs in the context of distributed name systems and highlight their relationship to GADS. Additionally, we provide some technical background on Distributed Hash Tables, a central component in our distributed design. We describe GADS's design in Chapter 3. In Chapter 4 we provide details on our reference implementation. Furthermore, we describe additional application-specific techniques that enable the use of GADS in the context of various common Internet applications, in particular, the WWW and e-mail. Our system exhibits certain benefits for Internet security in general; we discuss these in Chap-

ter 5. In this chapter we also present the results of a survey to answer some key questions about how GADS would perform in practice from the user's point of view. Finally, in Chapter 6, we review the results of this work and evaluate the contribution.

2. Related Work

In this chapter we will discuss two major fields of research that form the basis of this work. However, the first section deals with Rivest and Lampson's Simple Distributed Security Infrastructure (SDSI) design. The ideas in SDSI greatly influence the design of our name system. The second section describes the theory of name systems. The final section of the related work chapter is about the networking aspects that are required for our design goals, namely prominent examples of distributed hash tables and their properties.

2.1. Simple Distributed Security Infrastructure

SDSI, as defined by Rivest et al. [46], revolves around the idea of a simple public-key infrastructure (PKI) and linked local name spaces. The authors of SDSI claim that the present X.509 PKI is overly complex and incomplete because it relies on global name spaces. Three central design concepts of SDSI that are particularly important for our work are *principals*, *local name spaces* and *using certificates to create and assert name-value bindings*.

A principal in SDSI is a public key. SDSI has no notion of individuals or groups but only of principals. Since the principal is a public *signature verification* key, he can verify signed statements or requests made by other principals. SDSI allows any principal to create and sign certificates, effectively making all principals *Certification Authorities* (CAs).

Additionally, a principal can manage his own local name space, in which he can refer to other principals using arbitrarily chosen names. There is no global name space in SDSI that allows to uniquely identify principals by name. Instead, it is possible to link local name spaces by issuing name/value certificates. For example: If principal "dave" refers to another principal as "bob" in his local name space and "bob" refers to yet another principal as "alice" in his local name space, then "dave" can refer to the principal that "bob" calls "alice" as "bob's alice". Bob exports certificates asserting that his local name "alice" maps to a specific principal, making it possible to verify the mapping, if the principal "bob" is known and trusted.

The authors state that the process of creating a binding in a local name space from name to principal must be manual. The binding establishes a trust link from one principal to another and is an important process in the SDSI design. After all, judgment is required to validate that a specific public-key is actually owned by the individual or group that controls the principal. This is a process that all CAs have to do, like those in the X.509 PKI. The ownership validation processes and policies define the trustworthiness of the authority.

In SDSI, all principals are CAs and need to create initial bindings in their respective local name space manually. The authors claim that for the average user the creation of around 5-20 manual bindings is sufficient because linked name spaces allow principals to transitively follow bindings to foreign name spaces in a secure fashion by validating the exported certificates.

Principals can also bind other arbitrary values to their local names, making it an ideal concept for a secure name system. The linking of name spaces can be interpreted as secure subdomain delegation and principals as users controlling their own name space root. Referring to "bob's alice's www" can be easily transformed into a very familiar format: `www.alice.bob`.

The influence of SDSI on our design of a secure name system will become apparent in Chapter 3.

2.2. Name Systems

We use Zooko's triangle [60], an insightful hypothesis on the design space for name systems, to structure our discussion of related work:

Theorem 2.1 (Zooko's triangle)**.** *It is impossible to have a name system that achieves* **memorable**, **secure** *and* **global** *names at the same time.*

The adversary model for Zooko's triangle is the same as in GADS. Note that this model excludes a trusted third party acting as an authority. We now clarify the meaning of key terms in this conjecture.

Definition 2.2 (Memorable)**.** *A name is memorable if it is feasible for an attacker in our adversary model to obtain it by enumerating names (bit strings). In other words, the number of bits of entropy in the name is insufficient against enumeration attacks.*[1]

Definition 2.3 (Secure)**.** *A secure name system must enable benign participants to install and retrieve correct key-value mappings while experiencing active, malicious participants. The description of the adversary model for such participants is found in the introduction of this work.*

Definition 2.4 (Global)**.** *The system supports an unlimited, open number of participants without prior coordination or certification of participants. All benign participants receive the same global values for the same names.*

[1] We realize this appears to be a negative definition; however, the definition is meant to highlight the trade off between the simplicity of the name and the complexity needed to defeat enumeration attacks.

We have confirmed with Zooko Wilcox-O'Hearn, the creator of Zooko's triangle, that these definitions accurately represent the intended meaning of his formulation. On this basis we will now show as to why Zooko's triangle is a valid conjecture in our hypothesis:

Proof. As names are memorable, adversaries can enumerate all possible names. All participants, including the adversary, are able to add name-value mappings to the system. As the number of memorable names can be enumerated and the adversary can assume many identities, a powerful adversary can then add name-value mappings for *all* memorable names. This would prevent the assignment of any additional memorable names at some point due to names having to be global and secure. Thus, it is impossible to make a secure, global name system where memorable names are guaranteed to be available for registration by normal users without use of a trusted authority. □

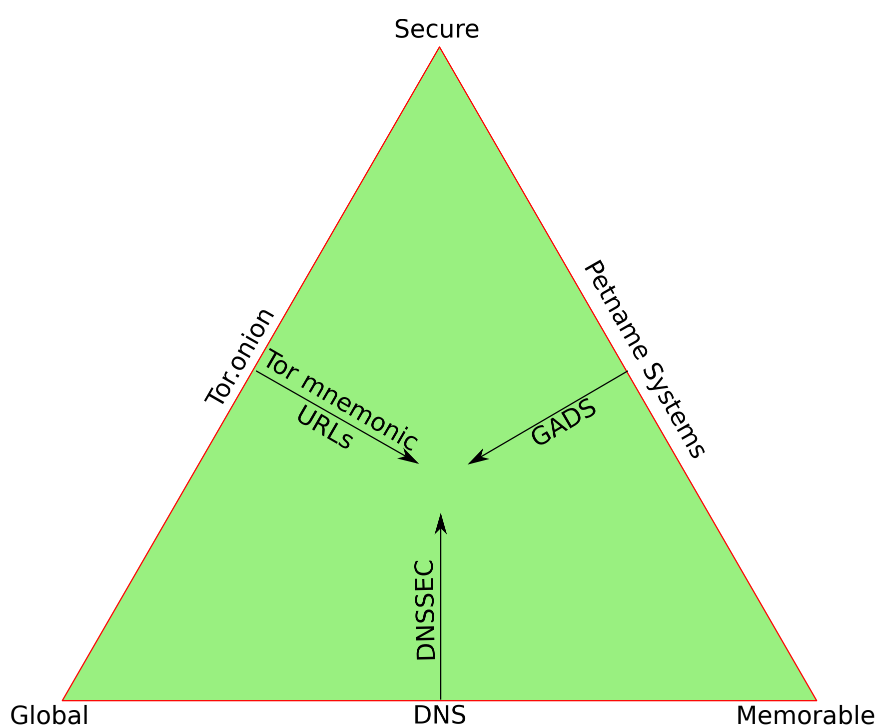


Figure 2.1.: Illustration of Zooko's triangle and key approaches to name systems in this context.

As a result any decentralized name system must compromise and choose a property to deemphasize. Figure 2.1 describes the three major design approaches in this context. We start with the edges of the triangle which represent the three simple (and extreme) designs:

DNS Globally unique and memorable names are managed by centralized organizations with no security guarantees [39].

Tor .onion Tor's ".onion" name space [11] uses bit strings derived from public keys as names, achieving strong security and global uniqueness at the expense of names not being memorable.

Petnames In a petname system, each user establishes a name of his choice for other entities [51]. Such a system can provide security and memorability, but the name-value mappings are not globally unique. A simple example of a petname system is the `/etc/hosts` file on UNIX systems.

Figure 2.1 also shows three attempts to move toward satisfying all three properties:

DNSSEC DNSSEC [2] is a set of security extensions for DNS. DNSSEC begins with a globally unique and memorable system (DNS) and improves security by adding integrity protection and data origin authentication. As trusted authorities remain, DNSSEC is not robust enough in our adversary model.

Tor mnemonic URLs The proposed Tor mnemonic URL system [48] begins with a secure and globally unique name system (".onion") and aims to make it memorable by encoding the hashes in ".onion" names into human-meaningful and memorable sentences. This is a mnemonic system. However, the resulting names will not be memorable by Definition 2.2 as the high entropy of the original names remains.

GADS The GNU Alternative Domain System presented in this paper begins with a secure and memorable petname system and makes the names transitive. This reduces the impact of not having global names. GADS proceeds to add a ".zkey" TLD which adds secure and globally unique names equivalent to Tor's ".onion" name space; however, this component is not vital to the system as such.

In order to be able to later highlight the differences between our design and existing systems, we will now give some additional background on some specific name systems that are particularly relevant to our work.

2.2.1. Petname Systems

A petname system is a name system where each user is in control of a *name set*. Each named entity in the set consists of three elements: a secure and globally unique *key* controlled by the entity, a local *petname* used by the owner of the name set to identify the entity, and a memorable but not globally unique *nickname*, which is chosen by the entity and used to generate a suggestion for the *petname*. The key idea behind petname systems is that users rarely have to deal with globally unique keys and instead usually deal with petnames: the petname system provides a mapping from petnames to keys. The nickname is an optional extension that enables users to suggest a name by which they want to be called. A user is free to replace the suggested nickname for an entity with a petname of his choice.

A simple real-world example of such a system is the so-called *buddy list* used in instant messenger applications. Users usually have one more-or-less cryptic name (i.e. the key)

that uniquely identifies the user in the system. Friends, however, may refer to a user by a petname. A petname is created when a friend adds a user's ID to his buddy list, e.g. "Jonathan Doe" with identifier "A174UX342" becomes "John". The instant messaging system may use the friend's real name (if given) as a nickname or to automatically derive a nickname. This nickname can then be suggested to the user for use as a petname. We describe how GADS uses nicknames in Section 3.2.1.

2.2.2. Domain Name System

The *Domain Name System* [40, 41] is a distributed and hierarchical name system. It is considered to be an essential part of the Internet because it provides mappings from *host names* to IP addresses. DNS is a distributed database of *records*. Each record consists of a name, type, value and expiration time.

Names in DNS consist of *labels* delimited by dots. Names are organized in a hierarchy. The root is the empty label, and the right-most label in a name is known as the top-level domain (TLD). Names with a common suffix (where the suffix must begin with the delimiting dot) are said to be in the same *domain*.

The *record type* specifies kind and function of the value that is associated with a name. In most cases, one name can be associated with many records of various types. The most common record types are "A" and "AAAA" records which provide IPv4 and IPv6 addresses respectively.

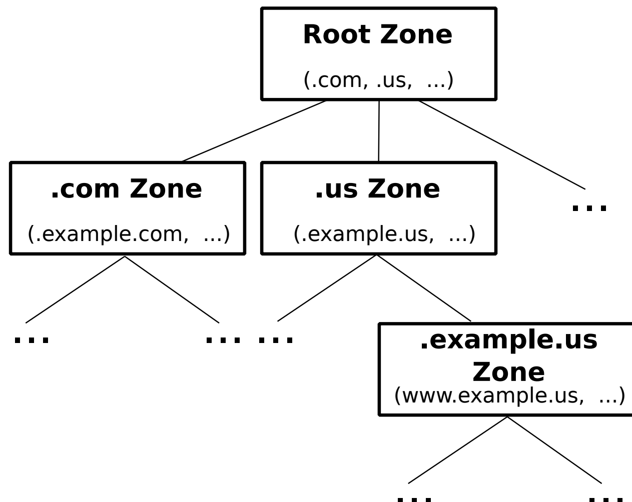


Figure 2.2.: Illustration of the DNS zone hierarchy.

The hierarchy of the name system enables the partitioning of the database into *zones*. A *zone* is a portion of the name space with a common suffix for the names (e.g. ".example.com") where the administrative responsibility is given to a particular authority. This authority has unrestricted autonomy over its domain; it is possible for an authority to delegate responsibility for particular *subdomains* to other authorities. This is done by adding a

"NS" record with the name of the subdomain. The value of the "NS" record then specifies the name of the DNS server which is the authority for the subdomain. Figure 2.2 illustrates the DNS zone hierarchy.

The *root zone* is the zone corresponding to the empty label. It is managed by the Internet Assigned Numbers Authority (IANA), which is in turn currently operated by the Internet Corporation for Assigned Names and Numbers (ICANN). However the National Telecommunications and Information Administration (NTIA) under control of the United States Department of Commerce assumes the legal authority over the root zone. This means that the content of the DNS root is controlled by ICANN. But any change to the root zone must be approved by the NTIA. The actual technical management of the DNS root zone and servers is done by VeriSign. This *multi-stakeholder* approach is wanted and endorsed by the US Department of Commerce to "ensure the long-term viability of the Internet as a force for innovation and economic growth."[2].

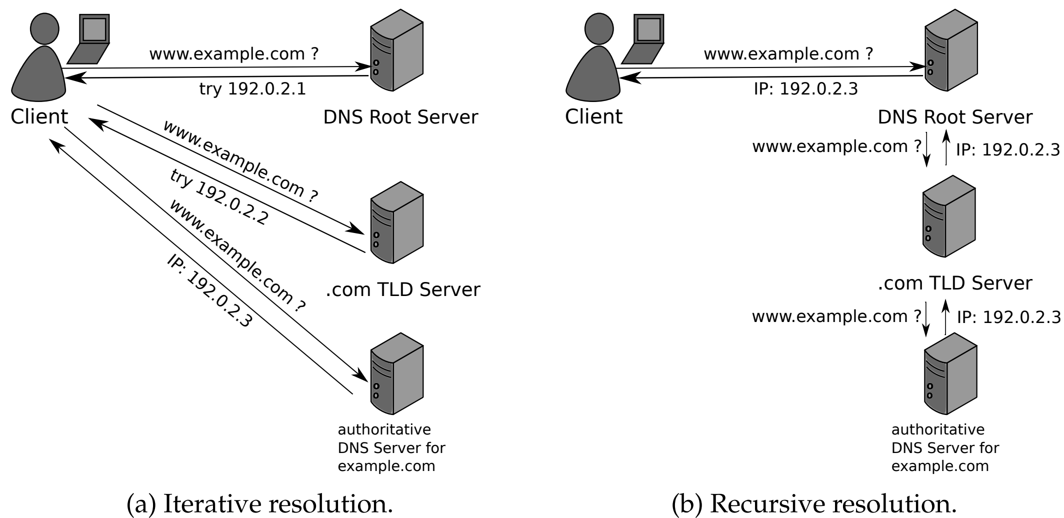
The root zone contains "NS" records which specify names for all authoritative DNS servers for all TLDs. For example, the "NS" records for ".com" may specify "*x*.gtld-servers.net" as the authority for ".com". Additionally, zones with "NS" delegations typically contain glue records (of type "A" or "AAAA") which provide a mapping of the names given in the "NS" records to the actual IP address for the authoritative DNS servers. TLDs are categorized into country code (ccTLD) and generic TLDs (gTLD) [44]. Country code TLDs are based on the official two letter abbreviations of the respective countries as defined in [26].

In DNS terms an *authoritative* DNS server for a particular zone answers queries for names mapped into this zone. DNS resolution can be done either *iteratively*, *recursively* or a combination of both. Whenever a client sends a recursive query to a DNS server it expects to receive a reply containing the value associated with the name or an indication that no value exists. Consequently, a DNS server receiving such a query either has to know the value associated with the name or forward the query to another DNS server. As a result the DNS server needs to maintain state information on all pending recursive queries. If the query is iterative the DNS server can send a reply containing an "NS" record to the client indicating that another DNS server is responsible for this name. In this case the client sends the original query again, this time to the DNS server specified in the previous reply. This process continues until a DNS server can actually answer the query or the authoritative server asserts that no record for the name exists.

DNS servers configured for a client system usually resolve iteratively because it allows them to cache results for other clients or repeated lookups. An example lookup for iterative and recursive resolution is illustrated in Figure 2.3. Caching is controlled by the expiration value that is part of each record. By using a timeout of days or even weeks, caching can significantly reduce the load and latency of the DNS. The authority must ensure that the mapping is valid until the timeout is reached, to provide consistency of the information despite caching.

In Figure 2.3a we can see that while iterative resolution potentially reduces the load of the root server because it does not have to resolve a name on behalf of the client, it is impossible to implement caching of the results (except on the client machine). On the

[2] `http://www.ntia.doc.gov/category/domain-name-system`, accessed 07/05/2012



(a) Iterative resolution. (b) Recursive resolution.

Figure 2.3.: DNS name resolution.

other hand, the recursive resolution technique, depicted in Figure 2.3b, allows the servers to cache answers resulting in very fast response times of frequently queried names, but all affected servers have to keep state for the pending lookups. In practice both resolution types are implemented: A so-called "stub resolver" is sending recursive queries to a DNS server that performs resolution on behalf of the client and employs extensive caching of results. In general, authoritative name servers, like the DNS root servers, do not support recursive resolution. Servers that only answer queries for names that the administrator manually configured are called "authoritative-only" DNS servers.

2.2.3. DNSSEC

The Domain Name System Security Extensions (DNSSEC) [2] extend the current DNS system with integrity protection and data origin authentication. DNSSEC does not provide transaction confidentiality or denial of service protection. DNSSEC introduces new record types for public keys ("DNSKEY") and for signatures on resource records ("RRSIG"). It relies on a public-key infrastructure in which all DNSSEC operators must participate. It establishes a trust chain from a zone's authoritative server to the trust anchor, which is associated with the root zone. This association is achieved out-of-band by distributing the root zone's public key e.g. in operating systems. As of July 15, 2010, a root zone trust anchor exists and root zone operators have started to deploy signatures and zone keys. The key of the root zone is held as a split secret between currently 21 trusted community representatives chosen by ICANN.[3]

Name resolution in DNS with DNSSEC is different from plain DNS resolution. We have to differentiate between a DNSSEC enabled "validating" stub resolver and a DNSSEC enabled "non-validating" stub resolver. A non-validating stub resolver must trust the replies of the queried DNS server. "Trust" here refers to both trusting the DNS server to correctly

[3] `http://www.root-dnssec.org/tcr/selection-2010/`, accessed 09/07/2012

verify the DNSSEC data and trusting the connection that is used to communicate. For example, the connection can be considered as trusted if IPSec is used [2]. Since the stub resolver is non-validating the only thing that is checked in the incoming DNS response is the "Authenticated Data" (AD) bit. In modern operating systems, stub resolvers either drop the non-authenticated responses or accept the response anyway and forward the results to the client. The APIs offered by operating systems are not designed to provide the user or program with authentication information and need to be changed appropriately for DNSSEC to be actually useful.

A "validating" stub resolver sends queries in recursive mode but additionally performs the signature verification itself. This results in end-to-end security for the client. Signature verification is performed by checking the signature in "RRSIG" records with the corresponding public key found in the "DNSKEY" record of the authoritative zone. "RRSIG" records are associated with RRsets – sets of Resource Records that share the same domain and type. To verify the public key, the resolver has to validate a corresponding "RRSIG" signature that signs the RRset that includes the "DNSKEY". This is done either by using the trust anchor or by following a delegation chain using "Delegation Signer (DS)" records. The "DS" record of a parent zone points to a "DNSKEY" record in the subzone and is part of a signed RRset in the parent zone. A resolver can follow the trust chain up to the trust anchor. "DS" records must be present for all "NS" delegation records where the DNS server is DNSSEC enabled and verifying.

The trust chains established by DNSSEC mirror the zone delegations of DNS. With TLD operators being typically subjected to the same jurisdiction as the domain operators in their zone, these trust chains are at a certain risk to legal attacks, especially where censorship is already established.

2.2.4. Tor .onion System

The Tor ".onion" System is used to access hidden services [11] in the Tor network. Names in the ".onion" name space are not resolvable by any DNS resolver because the ".onion" TLD is not part of the root zone. However, with the appropriate proxy software installed or using the "Tor Browser"[4], it is possible to access Tor hidden services using their ".onion" name. A ".onion" name consists of a 16 character alphanumeric hash followed by the ".onion" TLD. The label is generated by hashing the first 80 bit of the hidden service identity key using SHA-1 and encoding the result in base32. For example `https://jv6g2ucbhrjcnwgi.onion` could point to a valid hidden service.

It is obvious that such names are very hard to remember. But at the same time they provide strong security because Tor can use the identity key (in combination with the name) to authenticate the service. Basically the hash can be seen as the raw address of a Tor hidden service, much like an IP address. However, the ".onion" name also provides a mapping to the public key of the service adding strong security to the name system.

The Tor Project additionally offers *.onion web proxies* referred to as *tor2web* services. By using such a web proxy it is possible to use the ".onion" name space without having a

[4]`https://www.torproject.org/download/download.html.en`, accessed 08/14/2012

client proxy or Tor Browser installed. For example there is a ".onion" web proxy located at `http://tor2web.org`. To access the hidden service at `https://jv6g2ucbhrjcnwgi.onion` without having a local Tor installation on the client the user can simply point his browser to `https://jv6g2ucbhrjcnwgi.tor2web.org`. Of course this defeats the security assertions granted by the Tor software.[5] Tor2web services are DNS-to-Tor gateways that allow anyone to access Tor hidden services.

One particularly dangerous attack against ".onion" names are partial-match fuzzing attacks[6]. Imagine a user that regularly visits the service at `https://jv6g2ucbhrjcnwgi.onion`. An attacker could try to generate an identity key with a hash *similar* to the hash of the benign service. If the user encounters a link to the attackers service at `https://jv6g2uebnricnwgi.onion` he might falsely recognize the hash of the benign service and fall victim to a Man-in-the-Middle attack. This highlights the importance of memorable names for a secure name system.

Two proposals are currently discussed in the Tor community to make the ".onion" name system memorable. In the following we will discuss the *onion nym* and *mnemonic URL* system that try to solve the problem with different approaches.

2.2.4.1. onion nyms

By using *onion nyms* a hidden service administrator can register a memorable name for his `.onion` name at tor2web gateways. To do so the administrator adds a text file "onion.txt" into his hidden service web root[7]. Inside this file the desired name is specified. For instance "reg foobar" means that the hidden service would like to have the name "foobar".

When a tor2web gateway (for example `http://tor2web.org`) is used to access the hidden service it will look for the "onion.txt" file and retrieve the desired name. Unless the name is already taken the gateway will store a mapping from the memorable name "foobar" to the hidden service address `v2cbb2l4lsnpio4q.onion`. From then on it is possible to access the hidden service via the tor2web gateway by pointing the web browser to `http://foobar.tor2web.org`.

For this scheme to be feasible, all tor2web nodes need some kind of synchronization protocol to enforce the first-come-first-served policy and to ensure the consistency of the databases. This issue has not yet been addressed[8].

2.2.4.2. Mnemonic URLs

The idea of Tor mnemonic URLs is to take the hash of the `.onion` name and generate a human meaningful sentence that is easy to remember. The resulting sentence should be grammatically sound and contain a simple set of vocabulary. For example the name

[5] `http://tor2web.org`, accessed 08/14/2012
[6] `http://www.thc.org/papers/ffp.html`, accessed 09/07/2012
[7] Note that we assume here that the hidden service is in fact a web service
[8] `https://gitweb.torproject.org/torspec.git/blob/HEAD:/proposals/ideas/xxx-onion-nyms.txt`, accessed 09/07/2012

v2cbb2l4lsnpio4q.onion could be converted to the sentence "The quick brown fox jumps over the lazy dog", which is fairly simple and easy to remember. Client Software – like the Tor Browser – should make it easy for the user to convert from mnemonic to hash and vice versa. The actual URL should be generated by the software as well, following some rules to strip unnecessary minimal words. In our example the URL resulting from the mnemonic could look like this: https://quick-brown-fox-jumps-lazy-dog.onion. This scheme is less susceptible to partial-match fuzzing attacks than the original .onion names.

Since there is no implementation of mnemonic URLs at this point in time it is difficult to thoroughly evaluate the feasibility of this idea. However, there are definitively some theoretical issues with this approach. First of all, non-native English speakers might have trouble understanding and memorizing the sentences, depending on their level of English. The authors of the proposal also mention that simple translation of the sentences to other languages will not work due to the different meanings of words and possibly incompatible grammar.

2.2.5. Timeline Systems

Recently, timeline-based systems in the style of Bitcoin [43] have been proposed as a way to create a global, secure and memorable name system [53]. Here, the idea is to create a single, globally accessible timeline of name registrations that is append-only. In the Namecoin system[9], a user needs to expend computational power on finding (partial) hash collisions in order to be able to append a new mapping. This is supposed to make it computationally infeasible to produce an alternative valid timeline. It also limits the rate of registrations.

A possible drawback of the Namecoin system is that it is non-trivial for a user to verify the entire history. It is also not entirely clear if a single global timeline can be managed in a distributed fashion and still handle the transaction rate that global use might require. The need to spend CPU cycles on a proof-of-work just to register a name is also questionable. This inter dependency of technological and economic aspects literally forces users to waste energy and possibly creates legal issues associated with alternative currencies [23, 27]. Finally, the problem of name squatting and the resulting unavailability of names is also ultimately not addressed by Namecoin, especially if registration costs are set to be lower than for traditional DNS.

Sovereign Keys [13] and Certificate Transparency [32] are related systems which also use timelines to persistently map a host name to a public key or certificate. A key difference is that these systems focus on providing additional security for SSL/TLS and not on DNS. As a result, they generally rely on DNS (or DNSSEC with DANE [3]) to establish initial trust in certificates.

[9] http://dot-bit.org/, accessed 09/07/2012

2.3. Distributed Hash Tables

Some approaches attempt to improve the availability of DNS data by storing it in a distributed hash table (DHT). The Cooperative Domain Name System (CoDoNS) [45] is one example. The project claims that their approach is more resilient to denial-of-service attacks and equipment failures and reduces latency to "less than a single network hop"[10]. CoDoNS preserves ownership over names and is thus in the same legal position as DNSSEC. If data is not available in the DHT, the information is fetched from the normal DNS.

The name system presented in this work also uses a DHT to store and look up names. However, it is not bound to a specific DHT. It can be used with any DHT and then inherits the advantages and disadvantages of the respective DHT in terms of network performance, censorship-resistance and fault-tolerance. For this reason, we do not provide experimental measurements in this thesis, as the expected network performance and resistance to attacks follows directly from the extensive body of literature ([5, 6, 8, 9, 17, 21, 22, 29, 35, 47, 52, 58, 59]), just as the performance of the cryptographic operations follows directly from performance studies on cryptographic primitives [10].

However, for a reference implementation it is necessary to choose or develop a DHT with suitable properties to achieve our goals. Among the DHTs with freely available implementations, we settled for the R^5N Byzantine fault-tolerant DHT [15] as it best meets our needs. However, other secure and censorship-resistant DHT designs such as Whanau [34] or X-Vine [38] could theoretically be used as well. R^5N has the advantage that it does not require the use of a social network, while possibly being less resistant to Sybil attacks.

It should be noted that building a DNS replacement on top of a DHT does not introduce a dependency on DNS: all of the mentioned DHTs function just fine without an underlying DNS system as DHT routing protocols typically directly exchange IP addresses and not (DNS) names. In the following sections we will present some of the mentioned DHT designs in more detail to justify our decision.

2.3.1. Whanau

Whanau's [34] primary goal is to provide a "Sybil"-proof DHT by using social connections between users. Distributed Hash Tables are often weak against the "Sybil" attack. When performing a Sybil attack an adversary tries to create a large number of Sybil (phony) nodes. By doing so an attacker is able to disrupt the routing and maintenance operations of basic DHT designs [12]. Whanau relies on a social network like DBLP or Flickr to bootstrap its routing tables and counter the attack.

2.3.1.1. Design

Social networks have the useful property that it is hard for an adversary to convince honest users to peer with him. The attacker usually has to fall back on social engineering strategies

[10] `http://www.cs.cornell.edu/people/egs/beehive/codons.php`, accessed 09/07/2012

to connect his Sybil nodes with honest peers, which is considered to be rather costly. Due to this fact, Sybil nodes will mostly be connected to other Sybil nodes and only a few "attack edges" [34] connect Sybils with honest nodes.

This anomaly in the social network graph can be identified by looking for a *sparse-cut* in the graph. In combination with the "fast-mixing" property of social networks Whanau tries to sample mostly honest nodes into its peers' routing tables. The sampling technique is simply a random walk. If the random walk is long enough, it will approach the stationary distribution [7] and the end of the walk will be a random node in the network, with a probability distribution proportional to its degree. Because routing tables are maintained in this fashion and not by asking other nodes to disclose their routes, it becomes unlikely that an adversary is able to flood an honest node's routing table with Sybils.

Whanau also tries to counter Sybil clustering attacks by implementing so-called "layered IDs" [34]. Honest nodes pick IDs uniformly in the ID space. When performing a clustering attack multiple Sybil nodes choose IDs very close to a specific key, possibly making it impossible for other peers to look up. In Whanau each peer has multiple IDs, one ID per "layer". A node "uses a random walk to choose a random key as its layer-0 ID" [34]. According to the authors this ensures that "[...] honest nodes' layer-0 IDs are distributed evenly over the keyspace" [34]. However, the authors do not make clear as to why this is the case. The layer-n IDs where $n > 0$ are chosen by randomly selecting an ID from the finger table in layer-$n-1$. If there is a clustering attack present in layer-0, honest nodes will mimic Sybil nodes in other layers. On lookup a random layer is chosen. In this layer the appropriate finger is queried for the key. This process is repeated until a successful lookup is performed or a timeout is reached. The clustering attack is mitigated if there are at least $O(\log n)$ layers for each node. Lookups use $O(1)$ bandwidth making them very cheap. On the other hand, routing tables are expensive to maintain and rather large with $\Omega(\sqrt{n} \log n)$ space.

2.3.1.2. Assessment

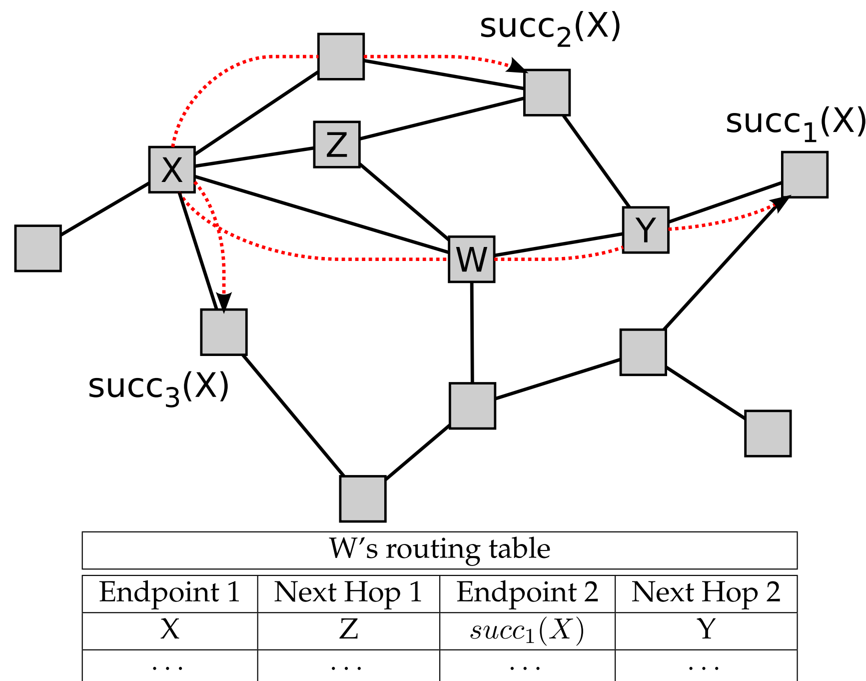
Whanau provides strong defense against Sybil and related attacks. At the same time it performs like other one-hop DHTs. Unfortunately, it can only handle moderate churn because rebuilding of routing tables is costly. Furthermore, the maintenance operations require nodes to expose their social connections to the network, a weak point in terms of privacy that is also mentioned by the authors of X-Vine [38].

2.3.2. X-Vine

X-Vine [38] builds upon the idea of Whanau to use a social network for DHT routing and Sybil resistance. What makes X-Vine special is that it embeds the DHT "directly into the social network fabric" [38].

2.3.2.1. Design

Nodes in X-Vine only communicate directly with their neighbors in the social network graph, not the DHT name space neighbors. This results in a system where information always has to follow a path in the social network graph and the set of records a node has stored serves as a routing table. On this basis, X-Vine provides pseudonymous communication, by only exposing a node's IP address to its neighbors in the social graph. Figure 2.4 illustrates X-Vine's design. A node in X-Vine is uniquely identified by a global and random numeric identifier. Additionally, it maintains a set of so-called "trails" in the network – paths to neighbors in the DHT name space. A trail consists of a number of "records" stored on each node along the trail. The information in a record includes the originating node and a destination node (the originating node's successor) as well as the next and previous hop along the trail.



W's routing table			
Endpoint 1	Next Hop 1	Endpoint 2	Next Hop 2
X	Z	$succ_1(X)$	Y
...	...	...	...

Figure 2.4.: An overview of the X-Vine overlay taken from [38]. It shows the social network graph and the friend relations of the nodes. The dotted paths are representing trails from node X to its successors in the name space overlay. The table shows node W's routing table based on this topology.

The properties of the social network graph also allow X-Vine to provide some resistance against Sybil attacks. This is done by limiting the number of trails that traverse a single edge in the social network. Since it is recognized that in a social network graph creating trust relationships with honest nodes is costly for an attacker, this limit will bound the number of Sybils in an honest node's routing table.

What distinguishes X-Vine from Whanau is the fact that its protocol does not require the users to expose their social network connections. This is a definitive improvement in terms of privacy. Also, due to the pseudonymous communication, it is not possible for an attacker to infer the communication between two nodes, because the mapping from IP address to DHT identifier is only known between directly connected nodes in the social network. To actually deduce the (direct) connection, and with it communication between two nodes, an attacker needs to have direct trust relationships with both parties.

2.3.2.2. Assessment

X-Vine provides a robust and performant DHT on the basis of a social network that provides decent protection against common attacks on routing and maintenance operations as well as Sybil attacks. However X-Vine also has some limitations. In particular, the hard requirement on an existing social network where a user's contact list can be used to bootstrap the DHT. The authors have provided proof of concept applications that prove the viability of the design but without the existence of a wide-spread *decentralized* social network this dependency is hard to meet at this point in time[11].

2.3.3. R^{5}N

R^5N is a Byzantine fault-tolerant DHT that works particularly well in restricted route topologies. Restricted route topologies contain nodes that are not able to directly connect to any other arbitrary node in the network. This can be due to Network Address Translation (NAT) middleboxes or the limited connectivity of friend-to-friend (F2F), mobile, sensor and wireless networks. The fact that peers can freely connect to each other is an imperative assumption made by most modern DHTs. Without this premise, the performance and reliability of DHTs like Kademlia can be diminished greatly, depending on the number of peers with impaired connectivity [15]. R^5N works around this issue by combining a random walk with recursive style Kademlia routing.

2.3.3.1. Design

The original Kademlia routing algorithm [22] follows an iterative approach. Whenever a peer performs an operation in Kademlia it tries to find *k* closest peers to a specific key. The distance metric used is XOR. First, the initiating peer queries the *k* closest neighbors in its routing table to find peers "closer" to the desired key. New, "closer" peers learned through those queries will be stored in the peers routing table. This lookup process is performed iteratively until no closer peers for a given key can be found. An inherent problem of this algorithm in restricted route topologies becomes obvious here: Peers closest to the key, discovered through iterative lookups, might be inaccessible for the initiating peer.

[11] The authors have mentioned Diaspora (`http://www.diaspora.com`) as a possible decentralized social network that could be used in the future.

R^5N is able to work around this issue by implementing a modified *recursive* Kademlia routing algorithm, at the expense of requiring more state. R^5N has a complexity of $O(\sqrt{n}\log n)$ bandwidth. Routing in R^5N is split into two phases. The first phase is a random walk of length $T \approx \log n$ where n is the number of peers in the network. This factor is derived from the mixing time of small world topologies that is also used by Whanau. The random walk ensures that the starting point of the second phase is a random peer in the network.

Phase two is a recursive style Kademlia protocol. It basically follows the standard Kademlia algorithm with the exception that routing is performed recursively: GET requests are forwarded to the closest neighbor in the routing table according to the XOR metric. PUT requests are forwarded in the same fashion unless the current peer is already one of the k closest peers, in which case the data is stored locally.

R^5N also employs a different strategy for routing replies back to the initiator. Because of path randomization each peer along the request path has to keep a list of active requests, resulting in a higher state requirement than other DHTs [15]. DHTs like Kademlia can use their normal routing algorithm to route replies making it unnecessary to keep a list of pending requests.

2.3.3.2. Assessment

R^5N does not provide the same protection against Sybil attacks like Whanau or X-Vine. However, it does not rely on a social network to bootstrap and maintain routing tables. Instead it is based on the established Kademlia algorithms for routing and routing table maintenance. Finally, the randomized routing algorithm and integrated DHT replication techniques provide excellent censorship-resistance.

In the end we settled for this DHT because it has the advantage that it does not require the use of a social network, at the risk of possibly being less resistant to Sybil attacks.

3. The GNU Alternative Domain System

This chapter is split into two major parts. We will first provide a detailed description of our name system, GADS, in Section 3.1. Thereafter, in Section 3.2, we discuss important aspects regarding the integration of GADS into modern applications and protocols.

3.1. Design of the Name System

GADS is fully decentralized. Instead of having a few US-based organizations manage the DNS root zone, a user runs his own zones. In particular, his own personal *master zone*. The user can create a GADS zone simply by generating a public-private key pair. Having the user in full control of his zones is the basis for censorship resistance in GADS, as an adversary cannot manipulate individually managed zone data.

In practice, GADS allows applications (and stub resolvers) to use DNS-like names by providing two TLDs, ".gads" and ".zkey". The ".gads" TLD always corresponds to the individual user's master zone. It is a secure, memorable, per-user name space. For ".gads", resolution results will depend on the individual user, because the user can freely assign memorable names in his personal master zone. Names in the ".gads" TLD are very similar to SDSI's linked name spaces. Example: `www.alice.bob.gads` refers to Bob's Alice's web server called "www". Each label in this name is a mapping in a GADS zone.

The ".zkey" TLD is an alternative to these context-dependent names. Names in ".zkey" are cryptographic identifiers and thus globally unique and secure names. The result of resolving requests in the ".zkey" TLD is independent of the individual user. However, names in the ".zkey" TLD are not memorable. The second label in a ".zkey" name is always the cryptographic identifier of a specific zone. All following labels are transitive zone delegations in the style of SDSI and names in the ".gads" TLD. Example: `www.alice.L1G6.zkey` refers to zone `L1G6`'s Alice's web server called "www".

A user can freely manage mappings for memorable names in his zones and delegate control over subdomains to other zones, including zones managed by other users (Section 3.1.1). The mappings of a zone are stored as *records* in a database on a machine under the control of the zone's owner. Records can be private or public. Public records are cryptographically signed with the user's private key and published in a DHT (Section 3.1.2). This allows name resolution and verification by other users. Record validity is established by the signatures and controlled using expiration values (Section 3.1.3). Consequently, secure zone delegation is possible by adding appropriate delegation records to a zone.

In the style of a petname system the user can create pseudonyms for his zones, specifying a preferred name. Users creating delegations into this zone can use the pseudonym or an arbitrary name in their local master zone.

Most of the record types known from DNS are also supported by GADS (Section 3.1.8). Additionally, GADS introduces new record types required for its operation. Section 3.2 provides more details on record types that are relevant for specific application scenarios.

3.1.1. Zone Delegation

Secure delegation of control over subdomains is central to GADS; it replaces the tree structure of DNS (with its inherent central point of attack) with a directed graph that is fully decentralized by nature. "PKEY" records have essentially the same purpose as "NS" records in traditional DNS. However, instead of delegating resolution for the subdomain to a nameserver specified by name, a "PKEY" record delegates resolution for the subdomain to the owner of the specified public-private key pair.

To realize this we use "PKEY" records that contain the *fingerprint* of the authority for a zone, which is the hash of the zone's public key. "PKEY" records delegate control over a subdomain to another zone. As names can consist of many labels, repeated delegation allows GADS to achieve transitivity of names. Figure 3.1 illustrates the use of "PKEY" records for delegation and name resolution.

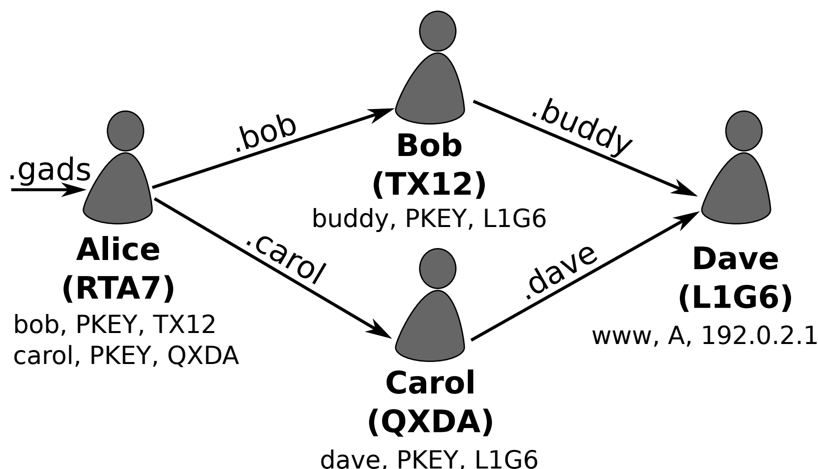


Figure 3.1.: Illustration of name resolution in GADS. Each user is shown with the fingerprint of the respective master zone and the public records from this zone (in the format *name, type, value*). The figure illustrates the paths Alice's GADS resolver would follow to resolve the "www.dave.carol.gads" and "www.buddy.bob.gads" names, which incidentally both refer to Dave's server at IP "192.0.2.1". Note that names in the ".gads" TLD are always relative; for Carol, Dave's server would simply be "www.dave.gads". While users are internally identified using public keys and records must be cryptographically signed, this is not visible in memorable ".gads" names.

3.1.2. Network Protocol and Routing

GADS uses a DHT as an efficient mechanism for obtaining records from zones that are managed by other users. For this, all public records for a given name are stored in a cryptographically signed block in the DHT under a key created by computing the XOR of the hash of the public key of the zone and the hash of the name.

To minimize the load on the network and to reduce latency, all validated records are cached in the local database. It is also the primary database for all of the zones for which the peer is authoritative (Figure 3.2).

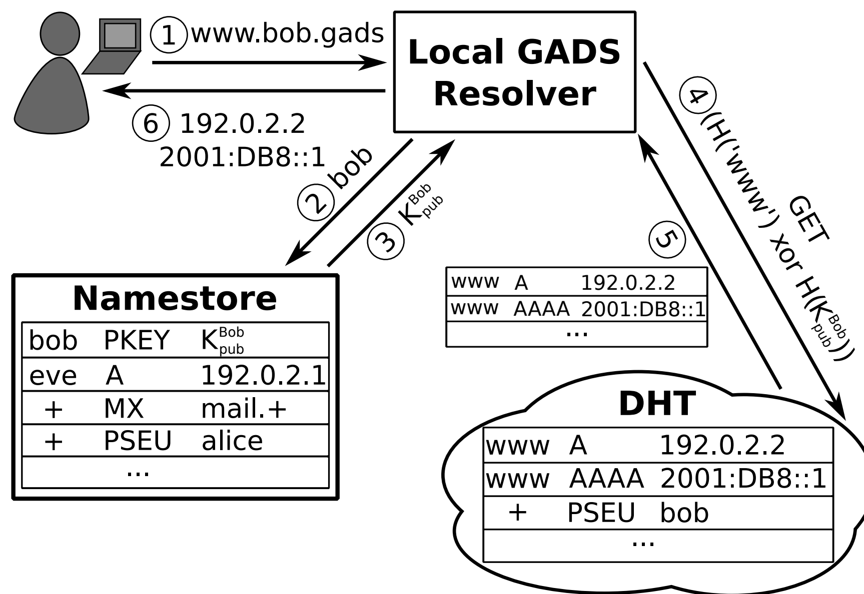


Figure 3.2.: In the GADS architecture, most requests will be answered by the local database which is the authority for all names in the user's personal ".gads" TLD. The user can delegate names to other users using "PKEY" records. If not cached, those names are then resolved using a DHT.

3.1.3. Signatures, Expiration and Freshness

To reduce the number of required signatures and in particular the number of signature validation operations, GADS always signs blocks containing the complete set of all records for the same name in the zone. For instance, if there is an "A" and a "AAAA" record for the name "www" in the local master zone, GADS will sign this set once with the zone's private key. Since additional records provided by resolvers often include additional records with the same name, this reduces the amount of cryptographic validation that has to be done.

Each record in GADS has an individual *expiration* time, which is freely determined by the authoritative zone's administrator. As in DNS, these expiration values are used to control caching.

In addition to the expiration values of individual records, the overall block in the DHT (which combines all records for a given name and the signature) also has a *freshness* time stamp. This per-name freshness time stamp controls how often GADS attempts to obtain up-to-date information from the respective zone's authority via the network. Specifically, the freshness time stamp determines the lifetime of blocks in the DHT, but not the lifetime of individual records in the cache of a local database. Signature expiration is also tied to the freshness time stamp of a block. When the freshness time stamp indicates that the signature is no longer fresh, all records associated with the record expire in the DHT — a fresh block needs to be published by the respective authority. However, individual records in the local database of an individual user persist. Here, the system will attempt to obtain a fresh mapping, but continue to use non-expired records from stale blocks to provide mappings to the user in the meantime.

Specifically, when GADS resolves a name, it first checks if a fresh block is available for the desired name and zone in the local database. If no block exists or the local block is stale, GADS attempts a DHT lookup to obtain a fresh block. While the lookup is pending, matching non-expired records that already exist in the local database (from now stale blocks) are used. Note that if a fresh block exists, a DHT lookup is not performed even if that block lacks a record of the appropriate type or is expired.

Generally, large expiration values for records should be used if possible to improve performance. Long expiration values allow peers to cache records longer and thus minimizes name resolution latency, as caching records enables the local GADS resolver to perform fewer DHT lookups. Record caching primarily benefits the user performing resolutions. The hosts running the authoritative peers for a particular zone do not really experience any significant load because record data is published in and resolved using the DHT and not on the host itself.

As with DNS, using large expiration values can make it impossible for administrators to quickly change the destination IP address for a service. Fast changes to records are sometimes needed for load-balancing or to direct users to a backup site in case of failures at the primary site. Another consideration for choosing expiration values is the possibility of individual authorities being offline; as we expect ordinary users to run GADS authorities, some of them will most likely be online only for a few hours each day. Caching and large expiration values can enable GADS to resolve names (especially as part of delegations) for which the authority is currently unavailable. We thus expect users to use long expiration values for delegations ("PKEY" records) and short expiration values for mappings to IP addresses ("A" records). This should work, as those users that do operate servers are likely to be online most of the time and can thus perform frequent republication, whereas users that do not operate servers will likely only use delegation to other authorities which will rarely, if ever, need to be updated.

A single DHT record block contains one or more records with the following wire format. The first field holds the 264 byte RSA Public Key Data including the Public Key of the records' authoritative zone. A 2048 bit RSA signature of the block signs all data following the *2048 bit RSA Signature* field. The first field of the signed data portion is used to specify the number of records in the block, followed by the name and actual record data in the *Records* field. *Records* is a variable length field containing *Record Count* records.

	0	8	16	24
0	RSA Public Key Data			
...				
65				
66	2048 bit RSA Signature			
...				
129				
130	Record Count			
131	0-Terminated Name			
...	Records			

This is the wire data format used for a single record:

	0	8	16	24
0	Expiration Time			
1				
2	Data Size			
3	Record Type			
4	Record Flags			
5	Record Data			
...				

A record contains the *Expiration Time*, the size of the payload *Data Size* (in bytes), the *Record Type* and *Record Flags*. Record flags specify metadata information regarding this record for internal use. For example, a flag can be set to make the record *private*, excluding it from DHT publication. All values for GADS specific record flags and types can be found in Appendix D for reference. The contents of the *Record Data* field are determined by the record type. Individual wire data formats of the supported record types can be found in Section 3.1.8.

3.1.4. Three Zones for Security, Privacy and Usability

The design presented so far suffers from a few usability issues, which are resolved in GADS by having three zones for each user instead of just the master zone. The use of these three zones is purely by convention (which is reinforced via the user interfaces); the GADS network protocol does not know about the three zones and additional zones could be created if necessary. In the following we present the three zones along with the issues they are meant to solve.

By default, a GADS installation uses the following three zones:

Master Zone: This zone contains all records and delegations that are available for the user himself as well as all other users. The user's master zone is where lookups start (".gads"). The master zone can contain entries to delegate to foreign zones as well as other resource records.

Private Zone: This zone contains all records and delegations that are confidential and that the user does not share. These records are not published in the DHT. While records can be individually marked as private and private records can exist in other zones, collecting all private records in a special zone makes it more obvious to the user as to which records are private and thus useless for sharing with other users. This zone appears by default in the master zone under the the name ".private.gads".

Shorten Zone: This zone is automatically populated by GADS to achieve short names. The shorten zone appears under ".shorten.gads" in the master zone.

We will now describe the use of the shorten zone in more detail. Name shortening in GADS addresses the problem of long delegation chains. If a user, say Victor, already has a name for a public key in his zone, name shortening will replace the long delegation chain with the existing name. For example, "alice.bob.dave.gads" might already be known to Victor as "alice.gads", in which case substituting "alice.bob.dave.gads" with the equivalent "alice.gads" simply improves readability.

Now suppose Victor's zone does not yet have a record for Alice's public key. In this case, Victor's resolver will consider Alice's pseudonym. Let this pseudonym be "carol". Victor's GADS resolver will then check if "carol" is already taken in Victor's *shorten zone*. If the name "carol" is available in Victor's shorten zone, Victor's system will automatically enter Alice's public key under "carol.shorten.gads". Naturally, Victor can disable this feature or later manually edit his zone to correct unfortunate choices that might be created by this first-encountered-first-assigned policy. Figure 3.3 illustrates the import algorithm.

Names created by this shortening mechanism are stored in the shorten zone and marked as private by default to ensure that Victor is aware of the automatic shortening process and to ensure that his name bindings are not accidentally leaked to the DHT by publishing shortened names.

3.1.5. Globally Unique and Secure Names

The ".zkey" TLD is used in GADS to provide a name space with globally unique and secure names. The label after the ".zkey" TLD must be the base32hex [28] encoded fingerprint of a zone. Names in the ".zkey" TLD are resolved by querying the respective zone for the name that follows the hash label. As each ".zkey" name uniquely identifies a public-private key pair, no authority is required to manage the ".zkey" TLD. It is possible to follow delegations created by "PKEY" records from within ".zkey" names. For example, with the zones from Figure 3.1, "www.dave.QXDA.zkey" would lookup "dave" in Carol's zone and then return "192.0.2.1" as the IP address for "www" in Dave's zone.

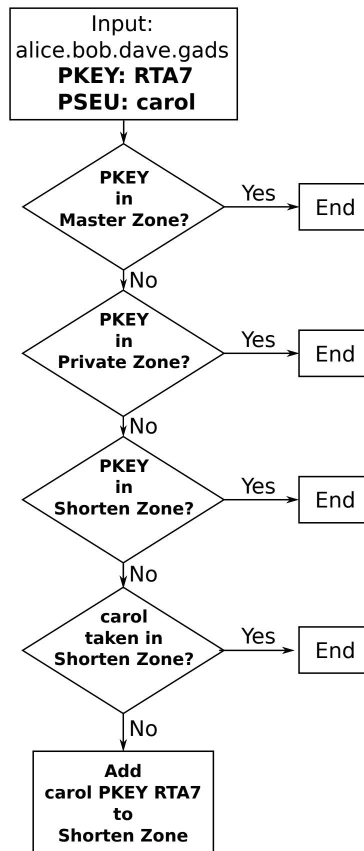


Figure 3.3.: A flowchart illustrating the GADS PKEY import logic. Automatic insertion of a PKEY record only happens if an equivalent delegation record cannot be found in any of the user's zones and the pseudonym name is not yet taken in the shorten zone.

The ".zkey" TLD is expected to be used under rare circumstances where globally unique names are required, and for integration with legacy systems (Section 3.2.3). Our user survey in Section 5.5 shows just how rarely the ".zkey" TLD would be required if GADS was widely used.

3.1.6. Zone Revocation

In case a zone key gets lost or compromised it is important that the key can be revoked. Whenever a user decides to revoke a zone and with it the zone key a revocation record of type "REV" must be published in that zone.

If the private key of a zone is compromised — but not lost — this record can simply be created on the fly and put into the zone in question. Any attacker in possession of the compromised key can publish records for this zone, but it is hard for him to remove the valid "REV" record from the censorship resistant DHT. In case the key is lost there needs

to be a backup of a signed "REV" record available that can be used for the revocation. GADS checks for the existence of a "REV" record when resolving "PKEY" delegations. Delegation will not happen if a revocation record is present.

3.1.7. Context dependent Names

In GADS a user can delegate to another zone using any name he likes. This results in a scenario where the authority of a zone cannot know what his zone is called by others. To solve this problem GADS assigns a special meaning to the label "+", which is used as a symbol for the zone of origin. This allows expressing that a particular name must be interpreted *relative* to the originating zone by ending it with ".+". For example, if a user receives a name of "www.+" from the "alice.gads" zone, the name is interpreted as referring to "www.alice.gads" from the user's perspective. A zone in GADS can also contain mappings with the name "+". It is used to identify records that are stored under the empty label. DNS has a similar semantic sometimes referred to as the "apex" of a zone. The apex of a DNS zone contains records under the name of the zone itself. For example the subdomain "example.com" might have an "A" record in its apex which will make it possible to access a service using "example.com" instead of "www.example.com". Some records types, like "CNAME"s are not allowed in the zone apex. DNSSEC also uses the apex of a zone to store the DNSKEY public keys. In GADS a user does not know what his zone is called by other people. But a user can use the label "+" to indicate that a record is in the apex of his zone. Note that "PKEY" and "CNAME" record types are not allowed in the apex of a GADS zone.

3.1.8. Names and Record Types

We now discuss the various record types supported by GADS in detail. To maximize backwards compatibility, GADS uses the same binary format and — with a few minor exceptions — identical semantics for the DNS resource records (RRs) that are defined in RFC 1035 [41] and RFC 3596 [54]. Furthermore, GADS defines a few additional GADS-specific record types; these records have record type numbers larger than 2^{16} to avoid conflicts with DNS record types.

The following DNS-compatible record types are currently supported by our implementation. Adding additional record types — especially existing ones from DNS — largely only requires writing the respective parsing and serialization routines.

A: Mapping of host name to IPv4 address. This record type works exactly as in DNS and it has the same wire data format:

	0	8	16	24
0	32 bit Address			

AAAA: Mapping of host name to IPv6 address. This record type works exactly as in DNS. The RFC specifies the following wire data format of the record:

	0	8	16	24
0	128 bit Address			
1				
2				
3				

CNAME: Mapping of a name to a canonical name. As specified in RFC 1035 [41], the query can (and in GADS will) be restarted using the specified “canonical name”. In GADS, the canonical name can be a relative name (ending in “+”), an absolute name (ending in “.zkey”) or a DNS name:

In the first case, GADS expands the relative name and continues the lookup with the result:

	Name	RRType	Value
Q:	www.example.gads	A	
A:	www.example.gads	CNAME	server.+
Q:	server.example.gads	A	
A:	server.example.gads	A	192.0.2.1

In the second case, we have to restart the query with the new name in the respective zone. For example:

	Name	RRType	Value
Q:	www.example.gads	A	
A:	www.example.gads	CNAME	hn.*hash*.zkey
Q:	hn.*hash*.zkey	A	
A:	hn.*hash*.zkey	A	192.0.2.1

Note: *hash* is a fingerprint of the public key (53-character base32hex encoded hash) in this table. In the third case, the second query and answer are exchanged with DNS instead of GADS:

	Name	RRType	Value
Q:	www.example.gads	A	
A:	www.example.gads	CNAME	www.ex.us
Q:	www.ex.us *(DNS)*	A	
A:	www.ex.us *(DNS)*	A	192.0.2.1

As with DNS, if there is a “CNAME” record for a name, there must not be any other records for the same name. This is the CNAME wire data format:

	0	8	16	24
0	0-Terminated CNAME			
...				

NS: In DNS, "NS" records delegate resolution for a subdomain. The value in the "NS" record is the name the respective DNS authority, and this name has a separate "A" record:

	0	8	16	24
0	0-Terminated NS Name			
...				

In GADS, "NS" delegation works slightly differently: "NS" records are used in GADS to delegate resolution of the GADS subdomain to a DNS zone. As a result, we need both a DNS name (to give to the DNS server for the resolution) as well as the IP address of the DNS authority. Thus, in GADS, a name with a "NS" record must also always have "A" (or "AAAA") records for the same name. The "NS" record tells us the name of the authoritative DNS zone of the DNS server and the "A" records provide the DNS server's IP addresses. For example:

	Name	RRType	Value
Q:	www.example.gads	A	
A:	example.gads	NS	ex.com
A:	example.gads	A	192.0.2.1
Q:	www.ex.com *(DNS)*	A	
A:	www.ex.com *(DNS)*	A	192.0.2.2

Given the first response, GADS will synthesize the DNS name "www.ex.com" from the "NS" record and the "www" remaining from the GADS name and send a DNS query to the DNS server at 192.0.2.1 based on the "A" record. The resolution then continues using DNS. Note that this record type enables delegation to *DNS* from within GADS. Delegation within GADS is done using "PKEY" records.

SRV, PTR and MX: These record types work the same way as in DNS, except that, like "CNAME" records, they can contain relative names which are expanded in the same manner as "CNAME" records. For example:

	Name	RRType	Value
Q:	example.gads	MX	
A:	example.gads	MX	mail.+
Q:	mail.example.gads	A	
A:	mail.example.gads	A	192.0.2.1

"SRV" and "PTR" records are post-processed in a similar fashion. The wire formats are identical in DNS and GADS.

MX:

	0	8	16	24
0	Preference			
1	0-Terminated MX Name			
...				

SRV:

	0	8	16	24
0	Priority		Weight	
1	Port			
2	0-Terminated SRV Target Name			
...				

PTR:

	0	8	16	24
0	0-Terminated PTR			
...				

SOA: GADS typically has little use for information from "SOA" records. However, "SOA" records might be useful for DNS-to-GADS gateways (Section 3.2.4) and can thus be stored within GADS. Relative names are again supported and expanded.

	0	8	16	24
0	0-Terminated MNAME			
...	0-Terminated RNAME			
	32 bit Serial			
	32 bit Refresh			
	32 bit Expire			

TXT: Association of user-defined text with a name. This record type works exactly as in DNS. It's content depends on the location of the record in the name space:

	0	8	16	24
0	TXT Data			
...				

TLSA: This record is defined by the DNS-based Authentication of Names Entities (DANE) working group [3]. As with DNSSEC, the trust chain that is established by GADS can be used to validate X.509 certificates without a trusted authority. The record associates various certificate related data necessary to validate a server certificate. The "Certificate Usage" field provides details regarding the certificate data present in the record. Certificate usages define whether the given certificate is the trust anchor that needs to be used to verify the certificate given by a host or if it actually is the host certificate and must completely match the certificate provided by the host. On the other hand the "Selector" field specifies what parts of the certificate must match. It is possible to select that only the public keys must match or even the whole DER encoded certificate. The "Matching Type" field specifies the data format of the "Certificate Association Data". The certificate data can be either the certificate or just selected parts of it (see "Selector"). Furthermore the data can be hashed.

	0	8	16	24
0	Cert. Usage	Selector	Matching Type	
1	Certificate Association Data			
...				

In addition to supporting the above DNS-compatible record types, GADS defines the following new record types:

PKEY: "PKEY" records are used to delegate resolution to other GADS zones similar to "NS" records delegating resolution to authoritative DNS servers. A "PKEY" record maps a name to the SHA-256 hash of the public key of the subzone's authority. The respective target zone's records are then typically resolved by querying the DHT and the PKEY hash is used to calculate the query key. Section 3.1.1 contains more details on the purpose and usage of "PKEY" records.

	0	8	16	24
0	SHA-256 PKEY Data			
1				
2				
3				
4				
5				
6				
7				

PSEU: This record type is used to specify the desired *pseudonym* (or nickname) for a zone. "PSEU" records must be put under the name "+" into the apex of the respective zone. The value of the record consists of a single label with the 63-character limit from DNS. Sections 3.2.1 and 3.2.2 explain the use of "PSEU" records in detail.

	0	8	16	24
0	0-Terminated PSEUdonym			
...				

LEHO: This record type specifies the legacy (DNS) host name for a name in GADS. "LEHO" records are used to enable backwards-compatibility for virtual hosting and SSL certificate validation in combination with the client side proxy as explained in Section 3.2.2. For example:

	Name	RRType	Value
Q:	www.ex.gads	A	
A:	www.ex.gads	A	192.0.2.1
A:	www.ex.gads	LEHO	www.ex.com

The host located at `www.ex.gads` can also be referred to as `www.ex.com` in DNS. Typically a SSL host certificate would contain the LEHO and not the GADS name.

	0	8	16	24
0	0-Terminated LEgacy HOstname			
...				

REV: If a "REV" record is present the resolution will fail. "REV" records need to be put into the apex ("+") of the respective zone. For example:

	Name	RRType	Value
Q:	example.gads	REV	
A:	+	REV	NULL

Additional record types may be defined in the future. In particular, should a need arise to revisit the choice of ciphers in GADS, it is possible to change the cipher suite by adding support for a "PKEY2" record type which would again delegate to an authority based on it's public key, except this time using a different hash function or a different public key cryptosystem. Our current implementation uses RSA-2048, SHA-512 for signing and SHA-256 for generating the ".zkey" names and for keys in the DHT. SHA-256 is used as 512 bits cannot be encoded with 63 case-insensitive alpha-numerical characters for this is the DNS label length limitation.

GADS limits labels to 63 characters and names to a total of 253 characters including delimiters to maximize compatibility with DNS. GADS also follows RFC 3490 [16] for encoding internationalized domain names. As a result, existing input methods and APIs for name resolution with DNS will continue to work with GADS.

3.2. Integration with Legacy Applications

This section will discuss various application-specific issues that need to be addressed before deploying GADS. The focus is on supporting tools and infrastructure which is needed to improve the user's experience for important use-cases that a viable replacement for DNS needs to handle.

3.2.1. Surfing with Pseudonyms and Petnames

Suppose Alice runs a web server and a mail server and sets up her master zone using GADS. After the public-private key pair was generated, Alice creates a "PSEU" record where she states that her preferred pseudonym is "carol". For her web server she creates an appropriate "A" or "AAAA" record under the name "www". For mail, she sets an "MX" record using the name "+" (as with "PSEU" records, we use "+" for her own zone).

Now suppose we have a second user, Bob. He performs the same setup on his system, except that his preferred pseudonym is just "bob". Bob gets to know Alice in real life and obtains her public key. He then adds her to his zone by adding a "PKEY" record. Bob can choose any name for Alice's zone in *his* zone. Nevertheless, Bob's software will default to Alice's preferences and suggest "carol", as long as "carol" is not already taken. Bob can easily check if a name is taken by performing the respective query against his local database for his personal master zone. This is important as it gives Alice an *incentive* to pick a pseudonym that is sufficiently unique to be available among the users that would delegate to her zone.

By adding Alice's "PKEY" under "carol", Bob delegates queries to the "*.carol.gads" subdomain to Alice. Thus, from Bob's point of view, Alice's web server is "www.carol.gads" and Bob can now e-mail her at "*username*@carol.gads".

Now suppose Dave is Bob's friend. Dave has added a "PKEY" record for Bob under the name "buddy" — ignoring Bob's preference to be called "bob". Bob wants to put on his web page a link to Alice's web page. For Bob, Alice's website is "www.carol.gads" and for Dave, Bob's website is "www.buddy.gads". Due to delegation, Dave can access Alice's website under "www.carol.buddy.gads". However, Bob's website cannot contain that link: Bob may not even know that he is "buddy" for Dave — not to mention that the HTML of Bob's website should ideally be the same for all of Bob's visitors.

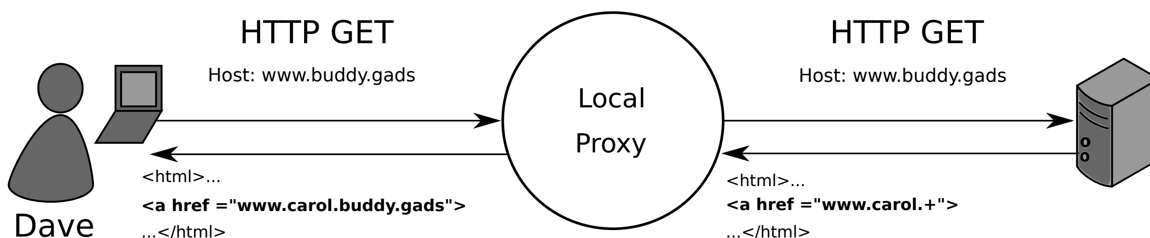


Figure 3.4.: Illustration of HTTP download with the GADS proxy. translating relative DNS names.

We solve this issue by having Bob use "www.carol.+" when linking to Alice's website. As before, the "+" stands for the originating zone. When Dave's client encounters "+" at the end of a domain name, it replaces "+" with the name of the GADS authority of the site of origin. When surfing, Dave's client would be his GADS-enabled browser, or a client-side HTTP proxy (Section 4.3.2) if the browser does not support GADS natively. Overall, this mechanism is equivalent to relative URLs [18], except that it works with host names. Once Dave's client translates "www.carol.+" to "www.carol.buddy.gads", Dave can resolve the name to Alice's public key (Figure 3.4) and eventually the "www" record in her zone.

3.2.2. Virtual Hosting and SSL Certificates

Virtual hosting (the practice of hosting multiple domains on the same IP address using HTTP 1.1) and SSL with X.509 certificates (which certify that a particular private key is used for a particular domain name) cause additional complications for any alternative name system.

The reason is that giving additional names to an existing service breaks a fundamental assumption of these protocols, which is that they are used on top of a name space with globally unique names. For example, a virtually hosted website may expect to see the HTTP header `Host: www.example.com` and the HTTP server will fail to return the correct site if the browser sends `Host: www.example.gads` instead.

Similarly, the browser will expect the certificate to contain the requested domain name "www.example.gads" and reject a certificate for "www.example.com" as this name does not match the browser's expectations.

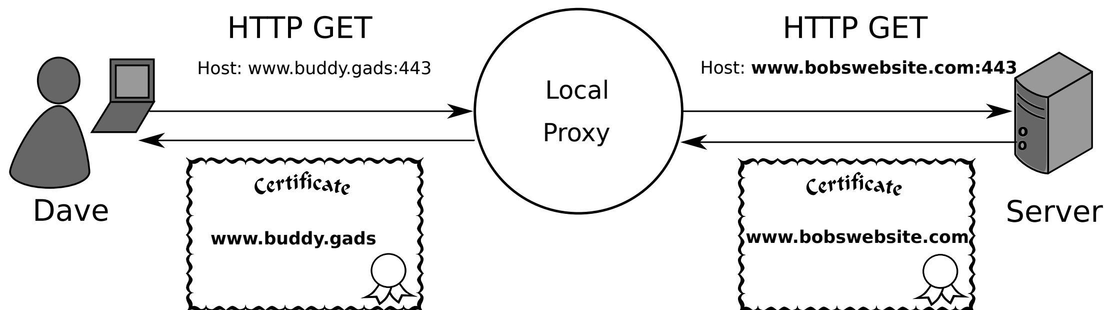


Figure 3.5.: Illustration of a HTTPS download with the GADS proxy validating legacy SSL certificates based on "LEHO" records and creating new certificates as expected by the browser on-the-fly.

Because with GADS each user is free to pick his own petname for the service, these problems cannot be solved by adding an additional alias to the HTTP server configuration or the SSL certificate. Our solution to this problem is to add a *le*gacy *ho*stname record type ("LEHO") for the name. This record type specifies that "www.example.gads" would like to be identified as "www.example.com". A proxy between the browser and the web server (or a GADS-enabled browser) can then use the name from this record in the `Host:` header or for SSL validation (Figure 3.5). Naturally, these are only legacy issues as the public-key infrastructure provided by GADS provides an alternative to X.509 certificates (see Section 5.1). Similarly, a new HTTP header with a hostname and a zone key could be introduced to address the virtual hosting problem.

3.2.3. Enabling Replies (e-mail)

Delegations in GADS create a directed graph among the zones. The resulting graph is typically not strongly connected. As a result, it is possible for Alice to have a name for Bob (i.e. "bob.gads") even if Bob has no such name for Alice. If in this situation Alice contacts Bob, she needs to supply him with her zone's fingerprint to enable Bob to respond.

The primary example for this scenario is e-mail. Fortunately, the necessary modifications are relatively simple. The sender would use a "Reply-to:" address with the fingerprint of his master zone ("username@*hash*.zkey"). The receiver can then use GADS to determine a memorable domain name for "*hash*", either by finding an existing name in the local database or by looking up the "PSEU" record for the zone and creating an appropriate entry in the local user's shorten zone. As a result, the ".zkey" name is only used in the network protocols and can be hidden from the users.

3.2.4. Accessing GADS without Installation

A *DNS-to-GADS gateway* is useful to allow legacy systems to access the GADS distributed database without installing GADS or changing their system configuration. This approach is similar to the tor2web gateways (Section 2.2.4) for the Tor network. A gateway will serve as a GADS resolver for a specific DNS suffix. The labels following the suffix will be interpreted as GADS names. Since names in GADS are not unique, a gateway can only serve specific zones or the globally unique ".zkey" name space. The GADS gateway behaves just like a regular DNS server. It answers queries for DNS names. However, if the resolution of a name in a specific zone is requested then the gateway will resolve the name using GADS. If the designated DNS-to-GADS zone of our gateway is `gads.com` then a query for `www.QXDA.gads.com` will be resolved in GADS. `QXDA` is referring to a GADS zone. The equivalent GADS name to query would be `www.QXDA.`*`zkey`*. We have registered a domain name in DNS (`zkey.eu`) where the DNS authority passes all requests on to GADS. For example, `www.`*`QXDA`*`.zkey.eu` would be resolved by the gateway querying the "www" record in the GADS zone `QXDA`. Anyone controlling a name in DNS can setup a DNS-to-GADS gateway. Figure 3.6 illustrates an example lookup of a GADS record using the DNS-to-GADS gateway.

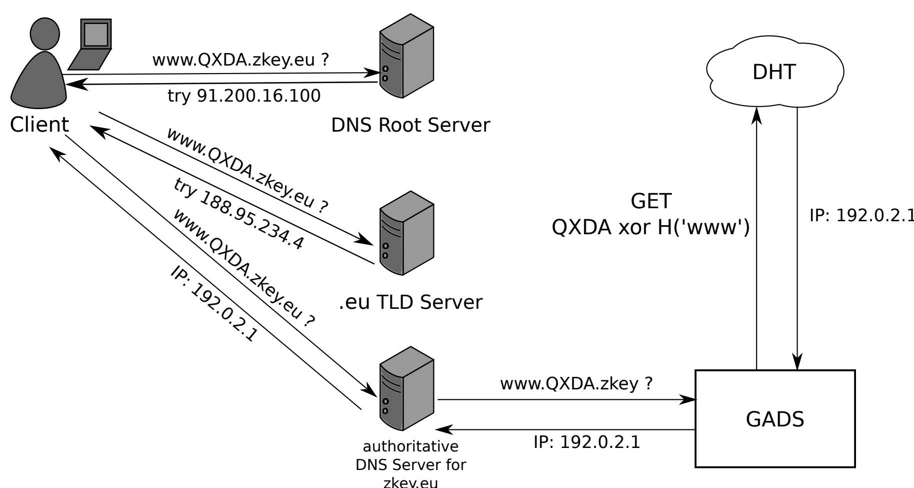


Figure 3.6.: Example lookup for GADS record `www.QXDA.zkey` by using our DNS-to-GADS gateway at `zkey.eu`. The IP addresses used in this figure are real world IP addresses pointing to an authoritative `.eu` DNS server (`a.nic.eu`) and our own authoritative DNS server (`toxic.net.in.tum.de`). Our server strips the name in the DNS query of the relevant parts (`zkey.eu`) and appends the ".zkey" TLD. The server queries the result in GADS and returns a DNS response.

The DNS-to-GADS gateway can also be used as a proxy DNS resolver for a local subnet. In this case the gateway will proxy all DNS request to an actual recursive DNS resolver. Clients are configured to use the DNS-to-GADS proxy as DNS server. This allows the client machines to send DNS queries to the gateway containing GADS names. All GADS names ending in ".gads" or ".zkey" will be resolved by the gateway in GADS. Normal DNS queries will be proxied to a recursive DNS resolver. Figure 3.7 shows such a setup.

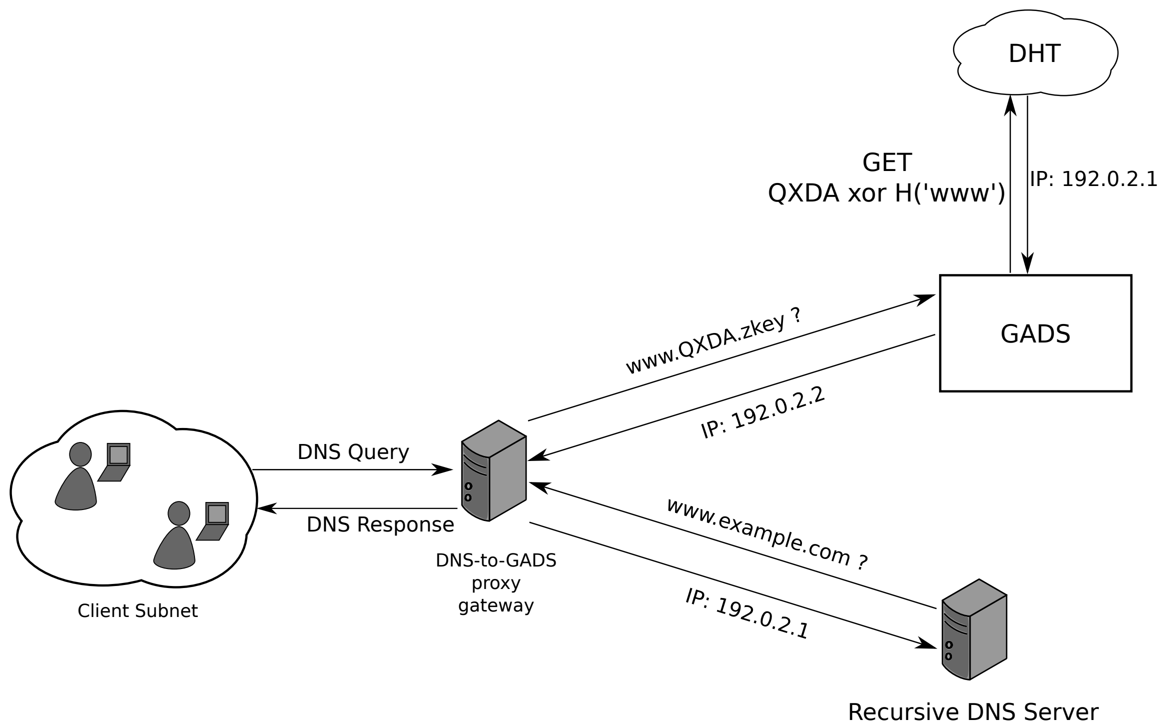


Figure 3.7.: A DNS-to-GADS gateway that is also a DNS proxy. A client using the gateway as its DNS server will be able to resolve names in GADS without having a GADS resolver installed.

While this trick can help users access GADS information without installing GADS, it does not offer any of the security or censorship resistance advantages of GADS, as DNS is used to access the proxy and thus DNS and the proxy operator would need to be trusted. This feature can only help users as a transition mechanism. For GADS to provide improved censorship resistance and security, users must install GADS locally and manage their own zone.

3.2.5. Incompatible Applications

In rare cases applications are completely incompatible with the GADS design. This results mostly from assumptions made on the network protocol of DNS. One such application is `Iodine` [1]. Iodine is used to tunnel IPv4 traffic in DNS packets. Basically, it encapsulates

[1] `http://code.kryo.se/iodine/`

IP packets in DNS queries and replies respectively. The Iodine client puts IP packets into DNS queries for a specific domain. The Iodine server is configured as an authoritative DNS server for that specific domain. The DNS query will eventually reach the Iodine server and it will extract the IP payload and forward it to its original destination specified in the IP header. The viability of DNS Tunneling in various practical contexts has been thoroughly investigated in respective works [57].

Figure 3.8 illustrates the scenario where the client is behind a firewall without direct access to the Internet.

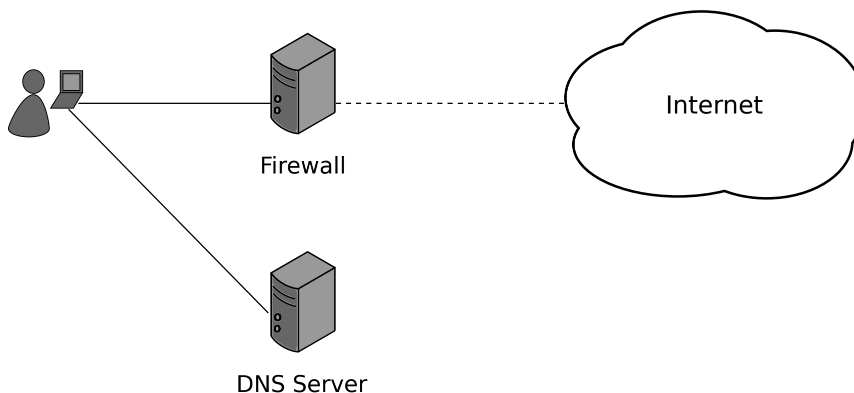


Figure 3.8.: The client is behind a firewall and unable to access the Internet. However, it is allowed to use a DNS server to resolve host names.

Using Iodine it is possible to bypass the firewall by tunneling all IP traffic via the DNS resolver. The dotted red line illustrates the tunnel that is established between Iodine client and server (Figure 3.9).

For GADS this approach cannot work because there is no such thing as an "authoritative GADS server" that could be used as the Iodine server. The authority in GADS is a specific peer. However, the query for the authoritative records is answered by the DHT and not the authority itself. Hence Iodine is one of those applications that are inherently incompatible with GADS and we cannot provide additional tools to "make it work".

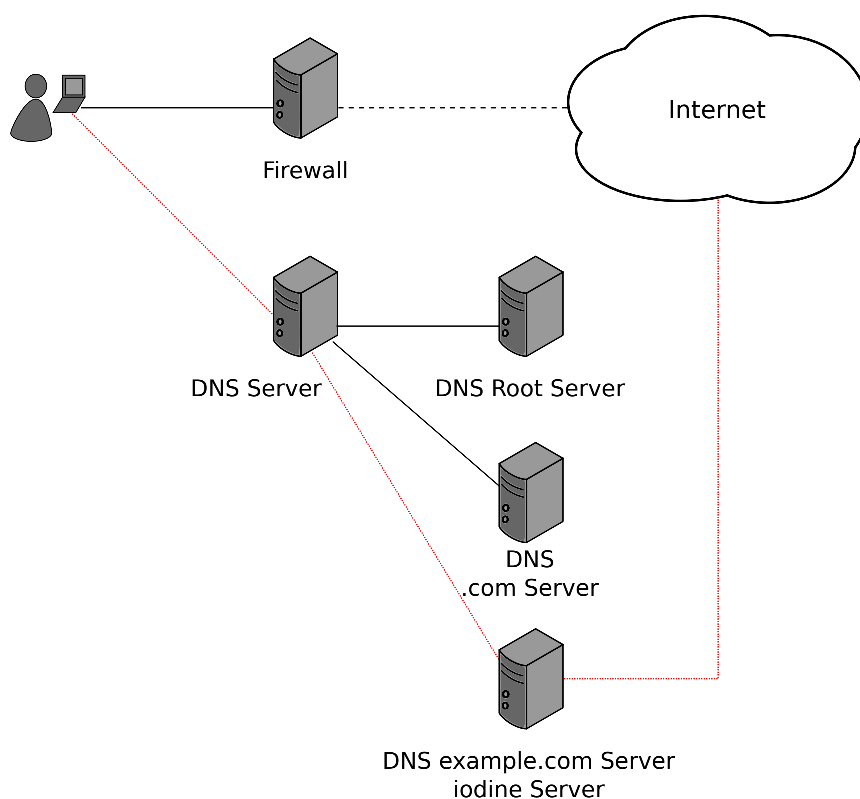


Figure 3.9.: The client uses Iodine to tunnel IP packets to the authoritative DNS server for "example.com". The authoritative DNS server is running an Iodine server instance and forwards the IP packets to their destination.

4. Implementation

In this chapter we present our reference implementation of the GADS design. Initially, we discuss how to integrate a DNS replacement into modern Operating Systems in Section 4.1. In Section 4.2, we provide implementation details on the *GNUnet Name System* (GNS), our GADS implementation based on the GNUnet peer-to-peer framework. Finally, in Section 4.3, we introduce a few useful tools to manage a local GNS installation and take a look at application integration in Section 4.4.

4.1. Integration into Operating Systems

Programs that are unaware of the existence and semantics of the GADS name space will inevitably try to resolve GADS names in the same way as DNS names. Consequently, a GADS implementation needs to intercept all DNS queries for the ".gads" and ".zkey" TLDs and inject appropriate responses. All other TLDs are forwarded to the traditional DNS system. Our current implementation provides three alternative methods to do so:

- On GNU systems, a plugin for the name services switch (NSS) [19] in GNU libc can be used to answer GADS queries before a DNS request is ever created. Mechanisms similar to NSS exist for other platforms. We also have an equivalent plugin working on Microsoft Windows.
- The resolver configuration (for example, `/etc/resolv.conf`) can be changed to point to an IP address (i.e. 127.0.0.1) with a modified DNS resolver. We have implemented a DNS-to-GADS gateway (Section 3.2.4) which resolves ".gads" and ".zkey" TLDs internally, and acts as a proxy for all other TLDs by passing those requests to an actual DNS server.
- Rules in a host-based firewall can be created to intercept and redirect DNS requests before they can leave the host.

All three methods have advantages and disadvantages. In the following sections we will provide details on how the different options are realized.

4.1.1. Firewall-based DNS Interception

A host-based firewall, like `iptables` in Linux, can be used to intercept all outgoing DNS requests, preventing applications from bypassing modifications to the operating system's stub resolver. The requests are redirected to a service that processes the DNS queries. Depending on the queried name in the DNS request the service can either use DNS or

GADS for name resolution. The result will be sent back to the application as if it was answered by the DNS server specified originally.

While this approach allows us to transparently resolve GADS names in DNS queries we cannot distinguish between users on the same host. A UDP packet with DNS payload does not tell us anything about the user that issued the query. Consequently, the GADS zone that is used for GADS queries is the same for any user on the system. On true multi-user systems this contradicts the idea of GADS where each user manages his own zones.

Figure 4.1 illustrates our implementation of this approach. Here, we use a virtual network interface (using TUN [30]) and configure the system's firewall to redirect all outgoing UDP traffic on port 53 to the TUN interface — except for outgoing UDP traffic from users in the `gads` group. The only process running in this group runs the DNS resolver of the GADS system.

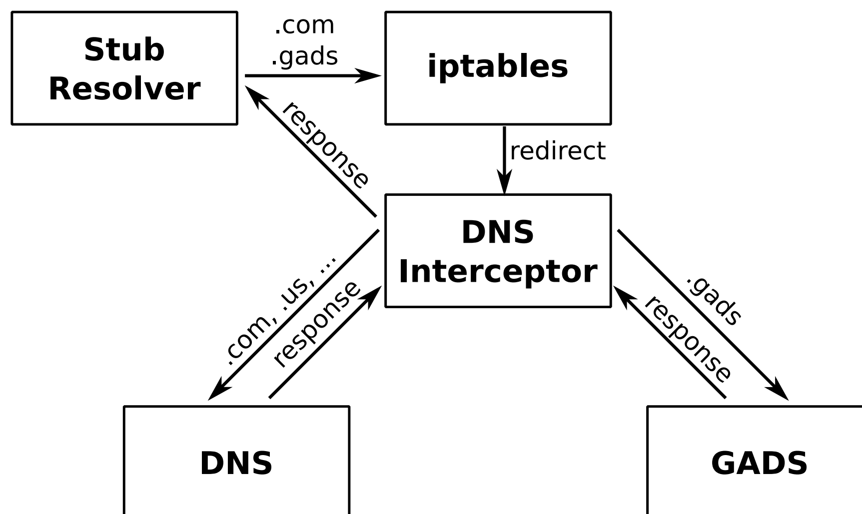


Figure 4.1.: Our GADS implementation uses `iptables` to intercept all DNS packets and appropriately handle GADS requests locally. Traditional DNS requests are proxied to an actual DNS resolver.

The GADS installation reads the DNS traffic from the TUN interface, resolves ".gads" and ".zkey"requests and forwards non-GADS requests to the original destination using the process running in the `gads` group (which is not affected by the firewall rule). In either case, the reply is sent back to the originating application via the TUN interface.

4.1.2. DNS-to-GADS Gateway

An alternative to the interception of DNS queries using the firewall is the configuration of an alternative DNS resolver which resolves requests to ".gads" and ".zkey" using GADS. Our implementation includes such a DNS-to-GADS proxy, which facilitates deployment of GADS to larger groups of users without the need to install a GADS resolver on each host.

Both network-level approaches, firewall interception and proxy gateways, limit the personalization of GADS requests to per-IP zones. However, GADS is designed with a per-user scenario in mind. This limits the utility of these setups to scenarios where users do not share hosts. Furthermore, if users use a local DNS-to-GADS proxy server, the local server would have to be trusted to perform the cryptographic verification. Finally, the DNS transfer between the host and the DNS-to-GADS proxy would not be cryptographically secured.

4.1.3. NSS Plugin

On GNU systems, the NSS-based approach has the key advantage that it allows our GADS implementation to learn the identity of the user that issued the query. As a result, we can fully personalize the GADS lookup on a per-user basis by maintaining a simple mapping between local user names and the respective zone keys. A potential disadvantage is that some applications may bypass the operating system functions and directly contact a DNS resolver. Tools such as `host` or `nslookup` do exactly that.

Whenever an application calls any of the `gethostbyname()` functions provided by glibc to resolve IP addresses, the name services switch is used. The name services switch consists of various plugins. Traditionally, at least one plugin reads a specific file for host information (`/etc/hosts`) and another plugin performs DNS queries. Our plugin is called `gns` in accordance with our GADS implementation discussed later in Section 4.2. A name services switch (NSS) plugin is usually configured in the file `/etc/nsswitch.conf` on a GNU/Linux system. An example configuration can look like this:

```
...
hosts:      files gns [NOTFOUND=return] dns
...
```

In this case the plugin `gns` will be asked before DNS to resolve a specific name. If our plugin is unable to resolve, it will return `NOTFOUND`. The string `[NOTFOUND=return]` tells the NSS system that it should return in this case and not ask DNS. This ensures that we do not leak the information of using GADS into DNS or the network in general whenever the resolution in GADS fails. Of course this does not affect non ".gads" or ".zkey" queries. In that case the GNS NSS plugin will always return `UNAVAIL` and the NSS will continue with the DNS plugin `dns`. NSS is the most suitable approach for GADS integration on multi-user systems. To also support applications that bypass the operating system resolver, one of the other two solutions can be used in parallel.

4.2. GNUnet Name System

GNUnet is a "framework for secure peer-to-peer networking"[1] released under the GNU General Public License version 3+. It is designed following a modular, layered structure consisting of various services. GNUnet implements the R^5N Distributed Hash Table discussed in Chapter 2 and provides us with all the necessary tools for a GADS implementation. The GADS service we implemented for the GNUnet framework is called the *GNUnet Name System* (GNS). It's main component is a GADS resolver. For data storage, cryptography, network operations and record handling GNS uses existing GNUnet services. We implemented the "Namestore" service for local record storage, signature creation and verification. For networking operations we use GNUnet's existing R^5N DHT implementation. The "VPN" service is used for "VPN" record processing. Figure 4.2 provides an overview of the design.

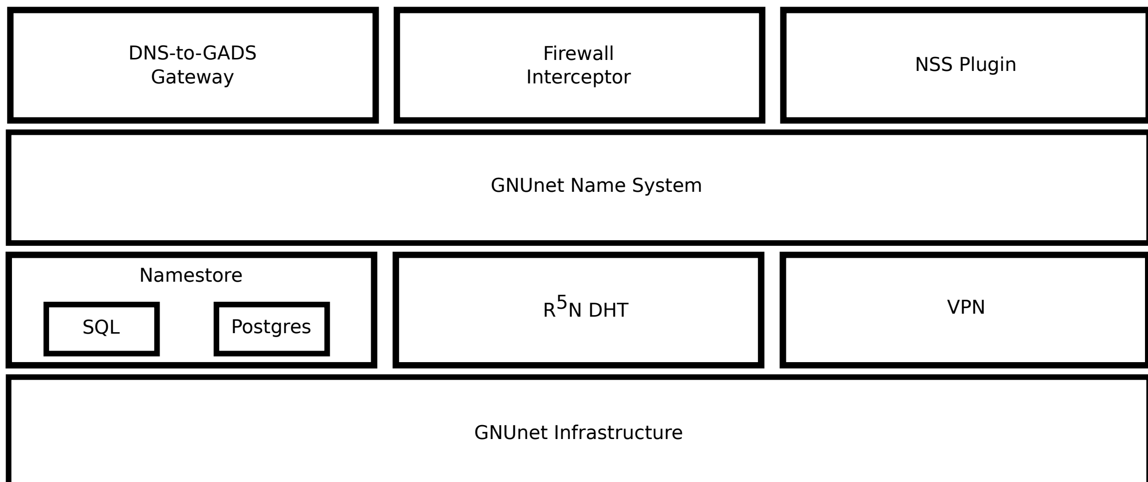


Figure 4.2.: GNS is built on top of the GNUnet services "Namestore", "DHT", "VPN". The Firewall interceptor, GADS Gateway and NSS plugin rely on the functionality provided by GNS.

In the following sections we provide implementation details on the before mentioned services.

4.2.1. The Namestore Implementation

The "Namestore" service provides a local database of GADS records. The API supports adding, removing and querying of GADS records as well as signature verification of GADS network blocks. In GNS this service is also used to cache lookup results from the network; Hence, the record data stored in the Namestore is not limited to authoritative records. Command line interface (CLI) tools are available to manage local records in the Namestore.

[1] `https://gnunet.org`, accessed 08/15/2012

However, there is also a graphical user interface discussed later in Section 4.3.3 allowing the user to manage the three GADS zone in a more usable manner. The Namestore service supports two different backends to store the record data either in MySQL or PostgresSQL databases and offers a plugin API that can be used to add support for arbitrary databases.

By design, the Namestore is in charge of creating signatures for authoritative GADS record blocks. Whenever an authoritative record is queried, the Namestore will create a signature for the block on the fly. For non-authoritative (i.e. cached) record data the signatures provided in the DHT blocks are stored and returned appropriately. For on the fly signature creation the signing key – the private key of the respective zone – needs to be online. Systems like DNSSEC provide offline signing of record data to provide additional security. In GNS, however, online signing is necessary. This is due to the relative expiration values of the records. Offline signing of records with absolute expiration values is no problem. Unfortunately whenever a user creates a record with a *relative* expiration value like "1 day" and the system publishes it into the DHT on the "1st of January 2012", the expiration value needs to be converted to the *absolute* value "2nd of January 2012". Since this converted value is signed data, the signing needs to be done right before the record is published into the DHT and not when the record is created. Consequently, the signing key has to be online for GNS.

4.2.2. Network Integration

Our implementation uses the R^5N DHT [15], a Byzantine fault-tolerant DHT that can conduct lookups with $O(\log n\sqrt{n})$ messages. For GADS, one of the interesting features of R^5N is the ability to do custom processing of replies within the network. In the GNUnet R^5N implementation this is done using *block plugins*. Block plugins are executed on every hop in the network for any "GET" request or "PUT" reply that is routed through the peer. There are various block plugins implemented in GNUnet for various different purposes. For example, the `fs` block plugin for file-sharing data or the `test` block plugin for arbitrary data. GNS uses our custom block plugin `gns`. Our custom processing logic in the `gns` block plugin verifies the signature for any reply that is routed through a peer. If the signature is invalid, the reply is dropped. This limits replies to properly signed data that matches the request. Figure 4.3 illustrates the block plugin logic. By dropping the reply before corrupt or incorrect data is forwarded to the initiating peer, the load on the DHT can be reduced. Our block plugin first checks if the incoming block is of the correct *block type*. The second check verifies that the key K_{Query} used to query for the record actually matches the zone information K_{pub} and record name $name$ in the block: $K_{Query} \equiv H(K_{pub}) \oplus H(name)$ where H is a cryptographic hash function. The third check makes sure that the record data is semantically correct and can be deserialized. Finally, after the signature has been successfully verified, we apply a Bloom filter [4] to identify and drop duplicate replies. The block is only considered valid and forwarded if all checks pass.

Another important function that is performed using the DHT service is record propagation. In GNS, this process is called *zone iteration*. As soon as the peer is started, all public records are put into the DHT. For R^5N it is advantageous to regularly perform "PUT" operations to achieve good balancing and replication of the data in the DHT. This also counters

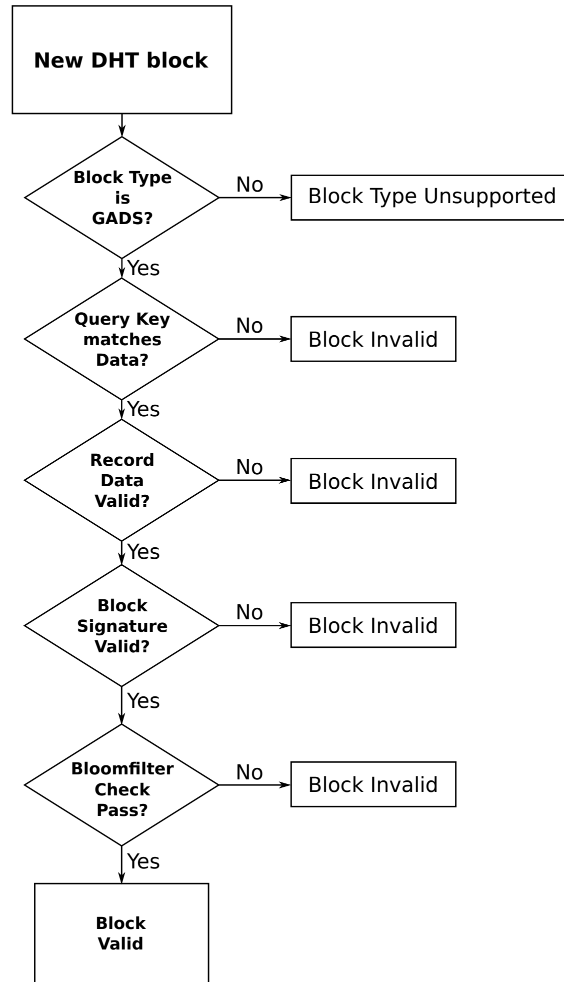


Figure 4.3.: The GNS block plugin logic.

data loss caused by block expiration. Intervals between "PUT" operations should not be too short or the zone iteration will cause heavy network load and performance on lookup will suffer. On the other hand very conservative record propagation intervals will cause zone iterations with a large number of records to take a long time. Consequently, static intervals, even if configurable, are not really a sensible option. The GNS service needs to determine dynamically the optimal interval for a zone iteration.

Counting all records in the Namestore can be extremely resource intensive if the number of records is very large. For this very reason the Namestore API does not even expose such a functionality. As a result, the number of records is unknown initially and GNS will on startup attempt to propagate all record data reasonably fast into the DHT. After this initial propagation an adjustment algorithm is used to adapt the interval to the record count.

All subsequent intervals between puts are calculated using a predefined initial "PUT" interval I_{init}^{PUT} divided by the current number of records $num_{current}^{records}$. The number of current

records is updated after each "PUT" accordingly. If the resulting interval is below a certain threshold I_{min}^{PUT}, the threshold value will be used as "PUT" interval instead. The interval $I_{current}^{PUT}$ for the next zone iteration is calculated using the following formula:

$$I_{current}^{PUT} = \max\left(\frac{I_{init}^{PUT}}{num_{current}^{records}}, I_{min}^{PUT}\right) \tag{4.1}$$

For the next zone iteration $I_{current}^{PUT}$ will be used as the time to wait between PUTs. The only problem left is that the Namestore database can be populated while a zone iteration is in progress. If the number of records increases by order of magnitudes then the zone iteration will take a very long time because the interval was calculated based on a much lower number of records. For instance, let us assume that $num_{current}^{records} = 5$ and $I_{init}^{PUT} = 5\ h$. The next zone iteration interval is calculated as follows:

$$I_{current}^{PUT} = \frac{5\ h}{5} = 1\ h$$

Furthermore, let us assume that after the first record is put, the user adds 45 more records to the Namestore. The zone iteration will not complete before $T_{total}^{PUT} = 49 \cdot 1\ h = 49\ h$. Not only will this drastically delay the next zone iteration, but it will also cause the new records to be available only after two days. To counteract this phenomenon the current PUT interval is adjusted while a zone iteration is in progress if the number of records $num_{current}^{records}$ that were already "PUT" in this zone iteration exceeds the total number of records $num_{last}^{records}$ counted in the previous zone iteration. In the example above after the 6th record is put into the DHT the interval $I_{current}^{PUT}$ is instantly adjusted using the Formula 4.1. Additionally, the interval is halved:

$$I_{current}^{PUT} = \frac{I_{init}^{PUT}}{2 \cdot num_{current}^{records}} \tag{4.2}$$

The total required time to put all 50 records can be calculated using the following formula:

$$T_{total}^{PUT} = \sum_{i=1}^{num_{last}^{records}} I_{current}^{PUT} + \sum_{i=num_{last}^{records}+1}^{num_{total}^{records}} \frac{I_{init}^{PUT}}{2 \cdot i} \tag{4.3}$$

$$= \sum_{i=1}^{num_{last}^{records}} I_{current}^{PUT} + \frac{I_{init}^{PUT}}{2} \cdot \left(H_{num_{total}^{records}} - H_{num_{last}^{records}+1}\right) \tag{4.4}$$

Where H_n is the *n-th* partial sum of the diverging harmonic series $H_n := \sum_{i=1}^{n} \frac{1}{i}$, also known as the *n-th harmonic number*. In our example the resulting total time for the zone iteration is:

$$T_{total}^{PUT} = \sum_{i=1}^{5} 1\ h + \sum_{i=5+1}^{50} \frac{5}{2 \cdot i}\ h \tag{4.5}$$

$$= 5\ h + 2.5\ h \cdot (H_{50} - H_5) \tag{4.6}$$

$$\approx 5\ h + 2.5\ h \cdot (4.5 - 2.3) \tag{4.7}$$

$$= 10.5\ h \tag{4.8}$$

Needless to say this is a significant improvement over the 2 days and 1 hour the zone iteration would have taken without this small adjustment. Of course simply setting the interval $I^{PUT}_{current}$ to a very low value like 1 minute or the default initial value I^{PUT}_{init} would reduce the duration even more. However, we think this is the better approach if we keep the properties of the DHT in terms of load balancing and performance in mind.

4.2.3. The VPN Service

GNUnet's VPN provides features such as IP tunneling and IPv4-to-IPv6 as well as IPv6-to-IPv4 protocol translation. In other words, each peer that is part of the VPN can provide access for other VPN peers to services and networks that might be otherwise unreachable.

Here is an example: Peers 0 to 4 are nodes in the GNUnet peer-to-peer network running the VPN service. Peer 2 is behind a NAT in the local subnet `192.168.0.1/24`. A web server with IP `192.168.0.1` is located in this subnet as well. Peer 2 decides to offer a VPN service with name "mywebservice" on port 80. If another peer wants to access the private web server in Peer 2's local subnet it has to establish a tunnel via Peer 2. Any peer can request a temporary IP address from the VPN service to Peer 2 by supplying the correct VPN service name ("mywebservice"), port (`80`) and peer ID (`2`). Figure 4.4 illustrates the discussed use case. Any packets sent by the requesting peer to the temporary IP address will be tunneled by Peer 2 to the private web server.

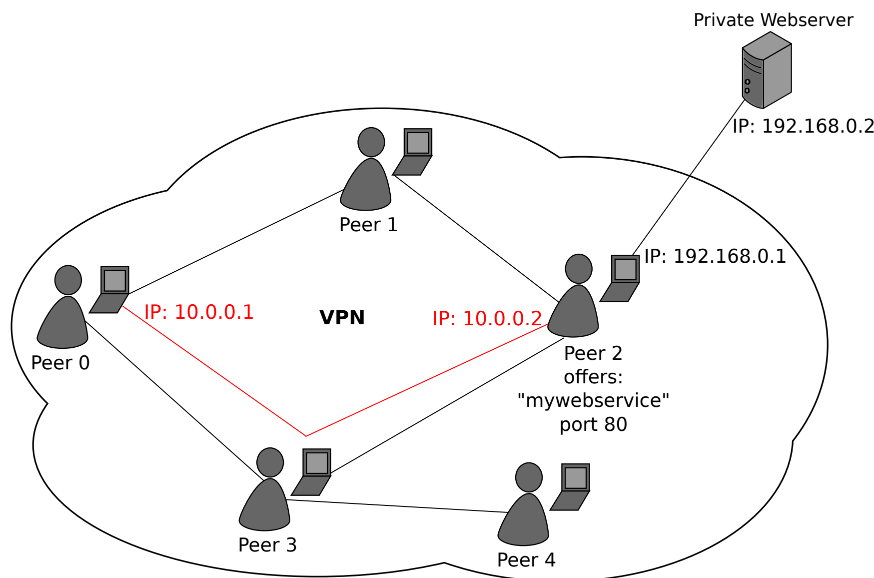


Figure 4.4.: Peer 0 requests a temporary IP address to access the service offered by Peer 2. The allocated IP addresses used by Peer 0 are 10.0.0.1 and 10.0.0.2. Peer 2 offers the service under the name "mywebservice" on port 80. The traffic received by Peer 2 is tunneled to a web server located in a private subnet that only Peer 2 has access to.

The required information to establish the VPN tunnel can be embedded into the "VPN" record type. Naturally, the record can be resolved by any application that uses the VPN functionality. Our GADS resolver supports on-the-fly "A" and "AAAA" record synthesis. Whenever the resolver encounters a "VPN" record and the queried record type is an IP address it will use the information in the "VPN" record to request a temporary address. The address will be put into a freshly created "A" or "AAAA" record that is returned to the application. Given that Peer 0 and 2 in our example are using GNS, the process of accessing the private web server becomes trivial. Peer 2 creates a record in its local GADS zone of type "VPN" under "www" containing the service name, port and the peer ID. If Peer 0 refers to this zone as "alice" it can access the private web server using the name `www.alice.gads`. GNS will transparently allocate an IP address whenever Peer 0 tries to access the web server using that name.

This is the VPN record data format:

<table>
<tr><th></th><th>0</th><th>8</th><th>16</th><th>24</th></tr>
<tr><td>0</td><td colspan="4" rowspan="8">SHA-256 GNUnet Peer Identity</td></tr>
<tr><td>1</td></tr>
<tr><td>2</td></tr>
<tr><td>3</td></tr>
<tr><td>4</td></tr>
<tr><td>5</td></tr>
<tr><td>6</td></tr>
<tr><td>7</td></tr>
<tr><td>8</td><td colspan="2">Port</td><td colspan="2" rowspan="3">0-Terminated VPN Service Name</td></tr>
<tr><td>9</td><td colspan="2" rowspan="2"></td></tr>
<tr><td>...</td></tr>
</table>

4.3. Complementary Tools and Programs

To assist the user in zone management and improve the usability we have created a few complementary programs. Mainly for headless systems like servers or for automated scripting GADS provides command-line tools (Section 4.3.1) to manage the local zone database and to query GADS names. As discussed in previous chapters, a proxy is necessary to deal with GADS names in HTTP sessions. In Section 4.3.2 we discuss a SOCKS proxy implementation for this purpose. Furthermore we provide a graphical user interface written in Gtk+. Section 4.3.3 introduces the program that ships with GNS.

4.3.1. Command-Line Tools

We have implemented two command-line tools that let the user manage the GADS zone and query the GADS system. The `gnunet-namestore` tool provides access to the Namestore databases. It queries the GNUnet Namestore service and requires a running service instance. To view all records in the root zone a user can issue:

```
$ gnunet-namestore --display
```

`gnunet-namestore` can also be used to add and remove records. These commands can be used to add an "A" record with the name `www` and the IP address `1.1.1.1` as well as a "LEHO" record pointing to `www.example.com`:

```
$ gnunet-namestore --add --type=A --name=www \
                   --value=1.1.1.1 --expiration="1 day"
$ gnunet-namestore --add --type=LEHO --name=www \
                   --value=www.example.com --expiration="1 day"
```

The expiration for the records is set to one day. Similarly records can be removed using the `--delete` switch. When adding records, the data provided with `--value` is checked for syntactic and semantic validity. A complete description of the `gnunet-namestore` program can be found in Appendix B.1.

A second tool called `gnunet-gns` is used to query and shorten names. Furthermore, it provides functionality to extract the authority of the name. `gnunet-gns` provides similar functionality to the `nslookup` and `host` programs found on most GNU/Linux systems. To lookup the previously added records for the name `www.gads` the user can use the `--lookup` switch:

```
$ gnunet-gns --lookup=www.gads
www.gads:
Got A record: 1.1.1.1
```

The record type for the lookup defaults to "A". If we want to lookup our "LEHO" record we can issue:

```
$ gnunet-gns --lookup=www.gads --type=LEHO
www.gads:
Got LEHO record: www.example.com
```

Another feature of the `gnunet-gns` program is name shortening. Given a name like `www.alice.dave.bob.gads` where "alice" is already in our root zone as "carol" we can use `gnunet-gns` to shorten the name:

```
$ gnunet-gns --shorten www.alice.dave.bob.gads
www.alice.dave.bob.gads shortened to www.carol.gads
```

This GNS feature is also used by the HTTP proxy discussed in the next section.

Finally, `gnunet-gns` allows us to determine the authority of a record. For instance, if the "www" label in the name `www.alice.dave.bob.gads` is an "A" record in alice's zone, then `alice.dave.bob.gads` is the authority:

```
$ gnunet-gns --authority www.alice.dave.bob.gads
alice.dave.bob.gads
```

Resolving the authority of a name is useful for relative links. The "+" in relative links will be replaced with the authority of the name. A more detailed description of `gnunet-gns` can be found in Appendix B.2.

4.3.2. HTTP Proxy

Our current implementation uses a client side proxy to do the expansion of relative names and SSL verification as explained in Section 3.2.2. A proxy implementation has the advantage that it works with virtually all browsers. The proxy speaks the SOCKS4a [33] protocol, which allows the browser to delegate resolution of domain names to the proxy. In the SOCKS4 protocol a browser usually sends the IP address and port of the desired connection to the proxy. If the proxy supports the SOCKS4a extension, the client can provide a domain name instead of an IP address. This is important as it allows the proxy to perform GADS resolution and obtain "LEHO" and "TLSA" records for certificate verification. If the target server is accessed using a GADS name, the proxy replaces relative GADS names in the HTML. Connections to systems using DNS names are simply proxied without processing the content. This logic is illustrated in Figure 4.5.

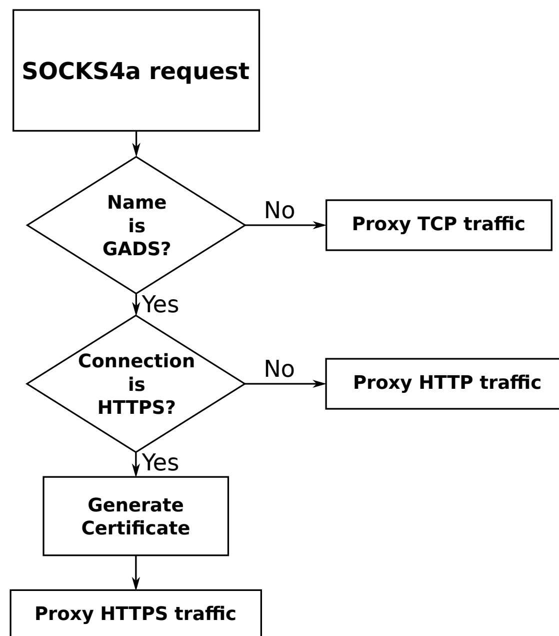


Figure 4.5.: Flow chart showing the connection setup of the proxy using the information provided in the SOCKS4a protocol.

Another issue the client proxy tackles is the *Same-Origin-Policy (SOP)* imposed by modern browsers. The SOP forbids scripts or cookies to access a different name in the do-

main name space. For example, if you browse `www.example.gads` then JavaScript code from `www.example.com` is forbidden to run. This can be an issue as the cookies and JavaScript code might use the legacy hostname (LEHO) instead of the GADS name and would then be ignored in accordance with the SOP. To solve this issue, the proxy translates links pointing to the LEHO and modifies the domain names in cookies to satisfy the SOP. This is implemented using HTTP header and HTML content rewriting and the use of Cross-Origin-Resource-Sharing (CORS) [56] headers.

For instance a “Set-Cookie” HTTP header field set by the server like this:

```
HTTP/1.1 200 OK
Content-type: text/html
Set-Cookie: Session=XXX; Domain=.example.com
...
```

where `example.com` is the LEHO for `example.gads` would be translated to:

```
HTTP/1.1 200 OK
Content-type: text/html
Set-Cookie: Session=XXX; Domain=.example.gads
...
```

In some rare cases the proxy cannot replace LEHO links, for example if a link is generated when the browser executes JavaScript code. As to not break the functionality of the web server, the following CORS headers are added as well:

```
HTTP/1.1 200 OK
Content-type: text/html
Set-Cookie: Session=XXX; Domain=.example.gads
Access-Control-Allow-Origin: http://example.com
...
```

This will allow the browser to execute scripts referred to using the LEHO. Note that the browser will use the DNS name in those cases and the GADS advantages will be gone. However, we think that it is more important to the user that the web page is working as expected.

The proxy also uses GADS domain name shortening as described in Section 3.1.4. GADS provides an API that shortens long delegation chains resulting in significantly shorter names. Note that the “PSEU” import and name shortening API are two complementary functionalities. The first is part of the GADS resolver and the basis for name shortening. The latter is the API that actually makes use of the automatically imported “PKEY”s.

For performance reasons shortening of names in the HTML is done only if the necessary information is already available in the local GADS cache. Otherwise, waiting for GADS to retrieve “PSEU” records for each link from the network would result in a significant increase in latency. If “PSEU” records are unavailable locally, the GNS service itself will

initiate a query for the respective authority in the background. The result of the background lookup will be cached in the local Namestore database; shortening will then be used the next time the name is encountered.

For web browsing it is necessary to use the proxy to be able to use functionality like virtual hosting and SSL. The proxy also needs to implement a trivial HTML rewrite engine so that users can follow relative links found on websites.

To generate certificates for GADS names we use gnutls [2]. If a connection request to the HTTPS port 443 is received the host certificate is checked using the LEHO by libcurl and a freshly generated GADS certificate is served to the browser in the SSL/TLS handshake. Essentially this results in a SSL-Man-in-the-Middle situation where the proxied data is available unencrypted to the proxy, similar to how sslsniff works [36]. This behavior is important though for two reasons. The first reason is that the certificate given by the host contains the DNS name and so the browser will not accept it. The other reason is that the GADS proxy needs to perform HTML and HTTP header rewriting, both tasks impossible if the content is encrypted. Since the proxy is meant to be operated by the user and run on the users machine, this does not have any security or privacy implications.

Figure 4.6 shows a screen shot of a browser visiting an ISOC web page with the GADS SOCKS proxy performing certificate validation based on the respective GADS records; the proxy generates a second certificate on-the-fly which is valid for the user's name for the site (here "myisoc.gads"). The browser accepts this certificate. In order for this to work, the proxy's signing key needs to be imported into the browser's certificate root store. The signing key is generated for each proxy instance on installation.

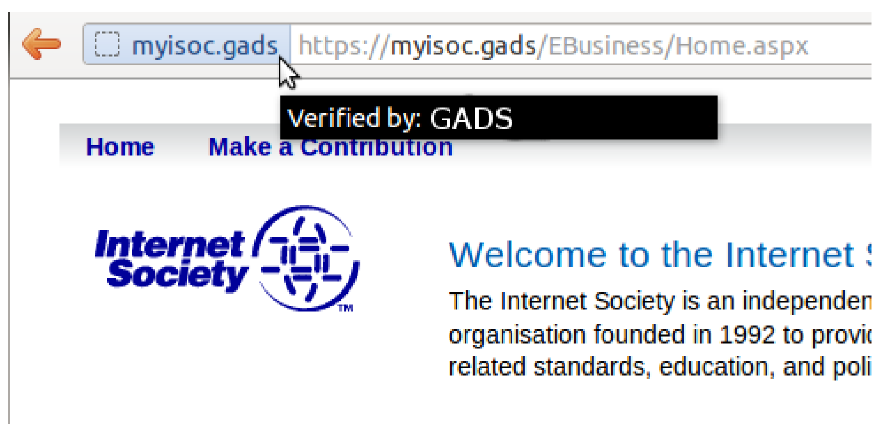


Figure 4.6.: Screen shot of a browser accessing. `https://myisoc.gads/` which is mapped by GADS to the host name and certificate of `https://portal.isoc.org/`. The GADS proxy validates the site's certificate and creates a certificate for the browser on-the-fly.

[2] `http://www.gnu.org/software/gnutls/`, accessed 09/07/2012

The implemented client side proxy consists of four major components:

- A SOCKS4a interface to communicate with modern browsers
- A GNU libmicrohttpd [3] component in charge of serving HTTP and HTTPS requests as well as to serve generated ".gads" certificates
- A GADS post-processor that processes the HTML content and headers
- libcurl [4] is used to access remote web servers via HTTP

Figure 4.7 illustrates the interaction of the various components that are part of the proxy.

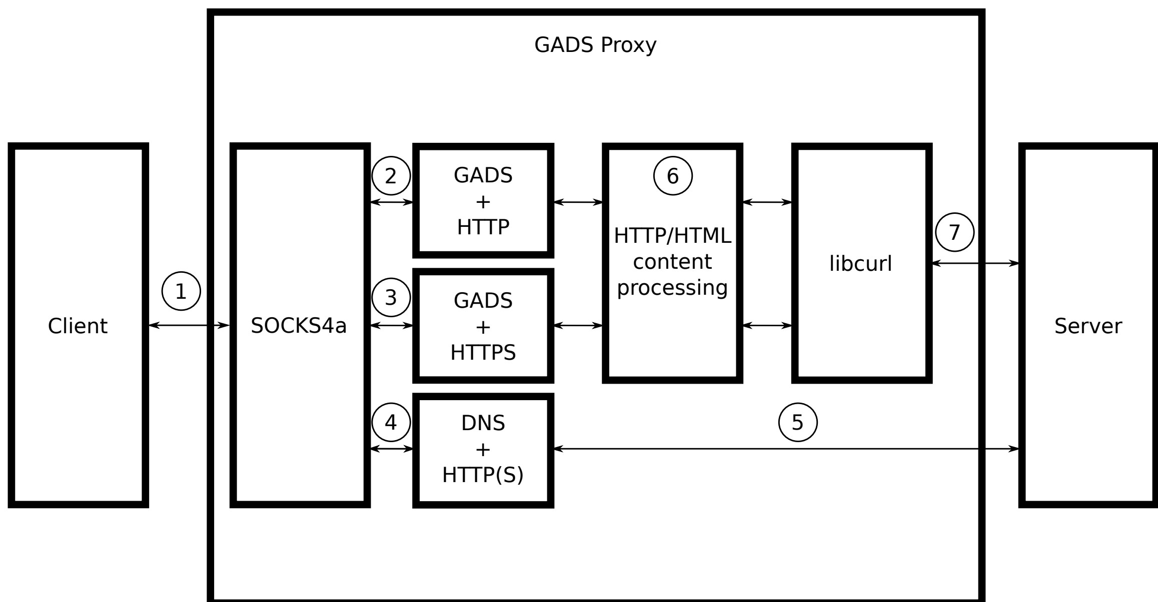


Figure 4.7.: This Figure illustrates the inner workings of the GADS proxy. The client uses the SOCKS4a protocol to connect to the proxy, providing host name and port of the desired connection (1). If the host name is in the GADS domain and the port is the standard HTTP port, the request will be handled by a GNU libmicrohttpd instance. If the port is HTTPS, the proxy will generate a SSL certificate for the client on the fly and a HTTPS GNU libmicrohttpd instance will handle the request using that certificate (3). In both cases the name will be resolved using the GADS resolver. If the host name is not in the GADS name space (4), the connection will simply be proxied to the desired server using DNS for name resolution (5). A GADS post processor modifies HTTP headers and HTML content to translate the GADS name to it's LEHO counterpart (if applicable) and vice versa (6). Finally, libcurl is used to handle the HTTP(S) communication with the server (7).

[3] `http://www.gnu.org/software/libmicrohttpd/`, accessed 09/07/2012
[4] `http://curl.haxx.se/libcurl/`, accessed 09/07/2012

4.3.3. The GADS Zone Editor and GADS QR codes

To give users the freedom GADS is intended to provide, it is important to allow the users to manage their GADS zones in a convenient way. We expect that most names will be learned from links as discussed in Section 5.5 and a few will be imported from out-of-band mechanisms manually. Still, users may want to create new names or manage the names that have been created by auto shortening. A screen shot of our GADS zone editor is shown in Figure 4.8.

Figure 4.8.: Screen shot of the GADS zone editor.

Our GADS zone editor allows users to create or delete DNS and GADS records in the master, private and shorten zones. It supports the creation of “A”, “AAAA”, “NS”, “CNAME”, “SOA”, “PTR”, “MX”, “TXT”, “TLSA” and “SRV” records and the GADS specific “PKEY”, “LEHO”, “REV”, “PSEU”, “VPN” records. The application performs an automatic syntactic and semantic validity check for the record data to prevent invalid records. The user can specify the desired validity duration for each record as an absolute (using a calender widget) or relative (“1 day”, “1 week”, “1 year”, “never”) value. Relative values are converted to absolute values upon publication in the DHT. Users can mark records in any of their zones as “public” or “private”, with the consequence that private records are invisible to other peers and will not be published in the DHT. Naturally, users are encouraged to place private records into their private zone to avoid accidentally using those names in links. The pseudonym for the user’s zone can be specified using a dedicated text input box.

To safely introduce direct trust relationships between GADS users we provide two different mechanisms. For one, the user can export his fingerprint to the clipboard to share it with other users out-of-band. Alternatively, we provide the possibility to create and export QR codes [1], which users can use to conveniently share keys in print, email or instant messengers. The QR code contains the fingerprint and desired pseudonym as a link of the form `gads://`*`hash`*`/pseudonym`. A QR code reader can recognize the QR code. The QR code reader would be configured to create a new name and delegation records in GADS

when it recognizes this type of URI. For example, such QR code could be put onto official mail by companies that is sent to their customers. Similarly, business cards could easily be fashioned with a QR code containing the GADS zone information of the company, person or both as well. Figure 4.9 shows the authors GADS QR code on a business card template.

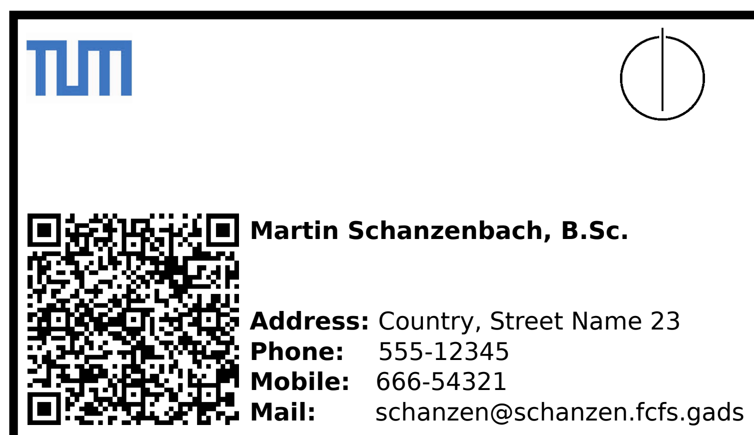


Figure 4.9.: A business card template featuring the QR code for the holders personal GADS zone.

4.4. Integration into Applications

To use the full potential of GADS a developer has three options. The first option is to use the DNS packet interceptor and manually create DNS queries. Alternatively, the GNS API can be used directly. The last option is to `fork-and-exec` the GNS command line tools discussed in Section 4.3.1. All three options have their advantages and disadvantages discussed below.

4.4.1. DNS packets

Creating DNS packets that are in turn intercepted by our Interceptor is probably the most straight forward solution for any application that already deals with DNS packets. The advantage is that the code needs little to no changes to resolve GADS names. A major disadvantage is that, as mentioned in Section 4.1, the intercepted query can only be associated with the system and not with a specific user. Hence, the important per-user zone management feature of the GADS system cannot be used on multi-user systems.

4.4.2. GNS API

The most common way of accessing the functionality of system services is through their libraries and APIs. GNS is part of the GNUnet framework and has a well defined API (Appendix C). The problem is that applications calling a GNUnet service API need to run in the GNUnet event loop. GNUnet features its own event loop, so that programmers can avoid the use of threads and services can be controlled in a simple and effective fashion.

Unfortunately, unless a programmer actually wants to write a GNUnet service or application, this dependence on GNUnet might be a hindrance. Programmers usually want full control over their applications, including main loop and threads. To avoid the GNUnet event loop, it is possible to use the CLI tools since they expose most of the functionality of the GNS API.

4.4.3. Fork-and-exec

`Fork-and-exec` is a common method for program interaction on Unix Systems. The `fork()` system call spawns a new program that is executed. Input and output of the spawned program (known as the `child`) are controlled by the spawning program, the `parent`.

Any program can execute the GNS command-line tools using `fork-and-exec` and gain access to the full functionality of the GNS API. The advantage of this method is that a programmer can entirely avoid linking against the GNUnet library. The GNS nsswitch plugin discussed in the beginning of this chapter uses `fork-and-exec` to invoke `gnunet-gns` and resolve GADS names.

Here is a stripped down example taken from the code:

```
1  FILE *p;
2  char *cmd;
3  char line[128];
4  struct in_addr ip;
5
6  if (-1 == asprintf (&cmd, "%s %s\n", "gnunet-gns -r -u", name))
7    return -1;
8  p = popen (cmd, "r");
9  free (cmd);
10 if ((NULL == p) ||
11     (NULL == fgets( line, sizeof (line), p )) ||
12     ('\n' != line[strlen (line) - 1]))
13   error ();
14
15 fclose (p);
16 line[strlen (line) - 1] = '\0';
```

```
17 if (1 != inet_pton (af, line, &ip))
18   error ();
19
20 return ip;
```

In this snippet the `popen()` system call is used to execute `gnunet-gns` and resolve the IP for the name in the variable `name`. In the subsequent lines the output of the command is parsed and if a well formed answer is received from GADS the IP is returned.

5. Discussion

In this chapter we discuss various specific security and usability issues associated with the migration from DNS to GADS. We show how GADS can be integrated into existing security infrastructures and reinforce their security assertions in Section 5.1. In Section 5.2 we highlight some security weaknesses related GADS name shortening and corresponding countermeasures. Section 5.3 discusses the bootstrapping process for GADS. The use of reverse proxies for host migration is outlined in Section 5.4. Section 5.5 presents the results of a small-scale survey into browsing behaviour. Finally, Section 5.6 discusses the viability of alternative GADS resolver implementations.

5.1. Establishing Trust with GADS

To replace X.509 Certificate Authorities (CAs), "TLSA" records in combination with GADS can be used, similar to IETF's DANE effort [3] for DNSSEC. Institutions with high security requirements can use GADS to establish more direct trust relationships with their clients, for example by furnishing them with the QR code of their respective GADS zone. The trust chain established by GADS allows the GADS proxy to use "TLSA" records to perform X.509 validation. As fewer entities with more direct relationships are involved, the resulting trust chains would be much harder to compromise.

The continued stream of security incidents with X.509 Certificate Authorities (CAs) [42] — which currently play a central role in securing SSL/TLS connections and thus transactions on the World Wide Web — has boosted the idea of reinforcing or even replacing these intermediaries. A central problem with current CA-based security is that any CA can create certificates for any domain name [25]. As a result, the security of the weakest CA determines the security of the system. With DANE, only the authority of the parent zone (which is typically the respective TLD) would be able to certify a given domain name.

Given the threat model of this paper, the TLD administration is a particularly bad choice as it is typically subject to the laws of the country under administration and will thus be a prime target for censorship and "lawful" intercept [31] activities. Trust anchors in distant parts of the world would be much less likely to comply with unethical requests as local laws might not apply to them.

It is conceivable that businesses might try to operate as trust anchors by offering a GADS zone with certain identity "guarantees" for their subdomains. In this context, GADS provides maximum trust agility [37], as users can freely determine their trust anchors and site operators can choose to be certified by a multitude of trust anchors, or establish direct relationships with individual users. Furthermore, the scope of a trust anchor is its subdomain and is thus limited and well-defined.

In terms of trust relationships, GADS resembles the PGP web of trust [55]. In fact, if name shortening is disabled, the user always receives rather explicit information about the trust graph, as the domain names then precisely correspond to trust chains in the web of trust.

5.2. Automated Name Shortening and Security

It is important that mappings that are created from automatic name shortening are placed into the special shorten zone. If this was not the case, an adversary might set his pseudonym to "bank" and automated shortening might then import this pseudonym into the zone of a user under "www.bank.gads", enabling phishing attacks. This restriction is particularly important as GADS supports internationalized domain names and thus the homograph attack [20] might be used to create names that look identical to those known to the user. Using the shorten zone limits this attack as the user can easily distinguish names ending in ".shorten.gads" from those he managed manually.

Name shortening not only generates more memorable names but also improves censorship resistance. Once Alice's record is added to Dave's zone, Bob can no longer sever the link. Furthermore, Alice can give her records a very long lifetime, resulting in Dave caching her information virtually indefinitely.

Name shortening also reinforces a user's incentive to pick a good pseudonym as a good pseudonym will dramatically increase the chance to appear under the same petname in all delegations. When we introduced "PSEU" records in Section 3.2.1, they were only used as a default suggestion when manually adding name mappings using public keys. Automatic assignment of names via name shortening is a much stronger method to establish a name, and thus creates a larger incentive to pick a unique pseudonym.

Furthermore, users that pick names for their pseudonyms which are too common are not only punished with long delegation chains for their names; they may also appear by default under names they would not choose for themselves. For example, suppose Alice obtains a link to Dave's website from Bob, who likes to refer to Dave as "freak" in his zone (this is a slander attack, equivalent to registering a slanderous name in DNS and pointing it to some victim's server). Alice's proxy would initially see "www.freak.+" as a link from "bob.gads" and translate it to "www.freak.bob.gads". This link is clearly not flattering for Dave, but shortening would usually automatically sanitize the name; however, if Dave failed to pick a sufficiently unique pseudonym, Bob's slander has a higher chance of being visible to Alice.

Shortening and relative domain names work nicely with search engines and other normal "surfing" activities where users click on links. For example, a search engine at "search.engine.gads" would generate a link "result.+" which would first be mapped to "result.search.engine.gads" and then likely be shortened to "result.shorten.gads".

5.3. Usability and Bootstrapping

One major problem we anticipate is that bootstrapping the use of GADS will be difficult as initially few GADS zones will exist and thus introduction via zone delegation will rarely be available. In order to give early adopters an immediate way to use the system without manually importing a large number of records into their personal zone, we have created a website where users can freely register names on a first-come-first-served basis for the "fcfs.gads" zone. The corresponding zone key is installed by default under the name "fcfs" in fresh GADS installations. We expect that various registrars with diverse registration policies will ultimately help with the bootstrapping problem by reducing the number of records users need to manually enter.

Even if the problem of providing users an initial set of names is solved, it is likely that GADS will initially be limited to a small set of administrators offering a rather small set of name data, especially when compared to DNS. To address this issue, we are considering using a zone transfer from legacy TLDs (such as ".com") to populate additional built-in zones with useful information. For example, a "2012-com.gads" zone could be created to preserve a snapshot of the 2012 ".com" TLD. Given such archived TLDs, DNS-level censorship would be easily circumvented (by using "evil.2012-com.gads" in case "evil.com" is censored in 2013). Naturally, the owner of the "2012-com.gads" zone key would need to be trusted, but the decentralized nature of GADS would make it easy to switch operators.

5.4. Improved Migration for Legacy Networks

Clearly migration to GADS cannot happen instantly; as a result, tools are need to enable a gradual migration of services and in particular websites to GADS during a phase where some users and some links may use GADS.

In particular, migration of websites can be facilitated using a reverse proxy that translates links in HTML pages from DNS names to GADS names depending on the name system supported by the client. This is crucial in the migration from DNS to GADS as during the transition period sites will need to work well with both name systems. In particular, DNS users need to be enabled to follow GADS links. One possible approach would be to use the "zkey.eu" migration mechanism described in Section 3.2.4. However, this would not result in readable links. Instead, the proposed solution is to use a reverse proxy to generate the appropriate output based on the capabilities of the client.

The capabilities of the client can be detected by looking for a special HTTP header which the GADS proxy (Section 3.2.2), or GADS-enabled browsers, can include whenever a HTTP request is transmitted to the server:

```
GET /index.html HTTP/1.1
Host: www.example.com
Gads: YES
...
```

If the reverse proxy encounters this `Gads` header, it should try to translate DNS names to GADS names. This can be done if the local GADS zone includes "LEHO" records matching the DNS names from the website. If no appropriate "LEHO" record is found, the DNS name can still be used.

On the other hand, if the HTTP header does NOT include a `Gads` header or a value indicating that GADS is not available, the reverse proxy needs to translate any GADS name (if there are any in the HTML) to a corresponding DNS name. This can again be done using the "LEHO" records for the given GADS names, replacing the GADS names with their "LEHO" values in the HTML. If there is no "LEHO" record for a GADS name, it cannot be replaced and DNS-only users will be unable to use the link.

5.5. Usability Evaluation: Surfing Behavior

Unlike DNS, the user's experience when using GADS depends on high-level user behavior: following a link corresponds to traversing the delegation graph and resolution is fully automatic. However, when users want to visit a fresh domain that is not discovered via a link, GADS requires a trust anchor to be supplied via a registrar or out-of-band mechanisms such as QR codes. This raises the question: how often are these inconvenient methods needed in practice?

To answer this question, we did a survey on surfing behavior. Specifically, we wanted to find out how often users would typically type in a new domain name for a site. A domain name is "new" if the user has never visited it before, and if the user is typing it in the name is also not easily available via some link. Typed in new domain names are thus the case where GADS would need to use some external mechanism to obtain the fingerprint of the zone.

Furthermore, we wanted to know how often users visit new domains via some link vs. visiting domains they visited before. This determines how often a GADS request can be satisfied from the local database vs. how often a network query (with possibly significantly higher latency) would be necessary. Combined, these two properties essentially determine the usability of our proposed system in terms of convenience (need to use out-of-band information) and performance (need to query the network).

Our method for answering these two questions exploits the fact that Firefox and Chrome keep a database with history information about the sites the user has visited. The database includes a flag that indicates if a name was typed in. To get access to this database, we asked friends, users of various mailing lists and visitors of our website to participate in a survey. We provided these volunteers with a simple shell script that would extract from the history database the number of URLs they visited, the number of unique domain names visited (which determines the expected size of a local GADS database) and the number of manually typed in unique domain names that were not previously visited via a link. Table 5.1 summarizes the results from our survey. The full table and the shell scripts can be found in Appendix E.

Table 5.1.: This table presents the representative results from our surfing behavior survey for a few representative users as well as the total over all 59 users that participated in our survey so far.

User	URLs visited	Unique domains visited	Fresh domains typed in	
1	57,651	4,313	274	(6.3%)
2	22,407	3,513	61	(1,7%)
3	13,696	1,836	179	(9,7%)
4	5,608	840	109	(13,0%)
5	1,210	576	22	(3,8%)
...	...	...	...	
Total	1,027,172	107,935	9,213	(8.5%)

Naturally, the browser history databases which were used in the survey are often incomplete as they are per-account, and include a finite view of the history as old entries are expired. Users can also purge their history manually. As a result, the numbers we obtained should be seen as *lower bounds*: if we had access to a longer history, it would become more likely that domain names that are currently classified as fresh and manually typed in might have been previously discovered via links. Figure 5.1 shows this negative correlation between the size of the history (in terms of number of sites visited) and the fraction of fresh domain names typed in. Thus, a GADS system which would over time have access to a much longer history can be expected to do even better than the data from Table 5.1 might suggest.

The survey shows that the number of entries in a typical user's GADS database will be at the order of tens of thousands of entries. This is rather small and thus good news, especially compared to timeline systems where the authors estimate that they would need to store about 190 million records and thus require between 16 GB and 300 GB of storage space [13, 53].

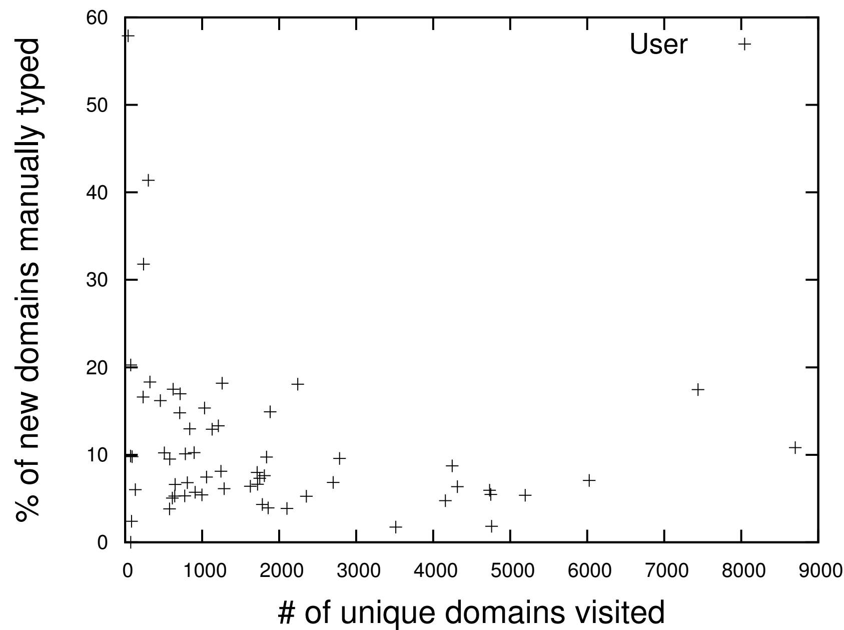


Figure 5.1.: This figure shows a (negative) correlation between the percentage. of new domain names typed in and the length of the available browsing history based on our user survey. Each point represents the data obtained from an individual browser profile.

Furthermore, the survey suggests that on average (for the limited timeline available in the browser's history) about 8% of the domain names are not obtained from following links and might thus require introduction using QR codes, registrars or even the ".zkey" TLD. While 8% is still somewhat high, it remains unclear how close this upper bound is to the actual behavior.

In terms of usability, this survey thus only partially answers the question of how often users would be forced to use the inconvenient (QR codes), insecure (registrars) or hard to memorize (".zkey" TLD) alternatives and how often we can expect users to use the desirable ".gads" TLD.

5.6. Alternative GADS resolvers

With GNS we have presented a GADS resolver implementation that uses the R^5N Distributed Hash Table. In theory, the GADS design could be implemented using other DHTs or frameworks. This, however, could be problematic in terms of compatibility and consistency.

Other implementations need store and retrieve record data from the same database. In the case of GADS this database is a DHT. If an implementation is using a different DHT

overlay, it will become incompatible to the reference implementation. Our implementation uses the DHT of the GNUnet peer-to-peer framework. Hence the alternative implementation needs to be built upon this framework as well.

Another more complex option is to bridge different DHT overlays. In this case, there needs to be at least one peer in the GNUnet network that bridges the R^5N overlay with whatever DHT the alternative implementation uses. The bridge will be connected to both overlays at the same time. This will inevitably influence both DHTs' properties because assumptions made by the routing algorithms are no longer valid.

If the alternative DHT operates independent of the GNUnet R^5N overlay, the name space is split, and depending on what implementation is used, only some name-value mappings are available. Since we can assume that zone ID's are unique, even beyond the DHT boundary, it is not a matter of name space collision, but of split-brain name spaces. A meta resolver could check all known GADS DHT overlays one by one. But this will result in high network load because of unnecessary queries and high latency in the resolution process.

Due to the above mentioned problems it is infeasible to implement an alternative, GNS compatible GADS system on top of a different DHT overlay. Alternative resolvers could be built on top of the existing DHT overlay, though.

6. Conclusion and Future Work

While DNS often provides globally unique names, there are many cases where this is not desired (i.e. for load balancing). DNS also cannot guarantee globally unique names as "trusted" DNS providers even today often change DNS results for technical, legal or adversarial reasons. However, what is important on the modern Internet is not so much global uniqueness but transitivity. We have demonstrated the use of delegation in a name system to achieve transitivity and are thus able to provide a censorship-resistant (secure), usable (memorable) replacement for DNS. We have explained how deploying GADS requires only minor modifications to existing applications as the system is largely suitable as a drop-in replacement for DNS. The overall design is deliberately simple, especially from the point of view of a user, as it needs to be understandable for the general public.

GADS is a censorship resistant name system which provides two types of names to users: nicknames which are memorable, secure and transitive, as well as cryptographic identifiers which are secure and global but not memorable. Placing names in the context of each individual user eliminates ownership, reduces their scope and thus the competition for names.

Large-scale critical infrastructure like DNS is never replaced overnight. However, GADS can be operated alongside DNS and begins to offer its advantages as soon as two parties using the system interact, enabling incremental, incentive-driven migration. For GADS adaptation, application and protocol designers may need to add support for context-dependent resolution of memorable names and for mapping globally unique cryptographic identifiers to memorable names.

The GNUnet Name System is part of GNUnet which is available to the public as free software under the GNU General Public License. The current implementation includes all of the features described in this work, except for a reverse proxy (Section 5.4) and integration with e-mail systems (Section 3.2.3). Future work also includes a tighter integration of GADS with user applications to increase user acceptance. For example, in addition to the proxy a browser plugin could be implemented. Furthermore, for name shortening and PKEY import the user could be notified by a desktop applet.

In the future, we will begin deployment to actual users and perform experiments to find out which usability problems arise with GADS.

Appendix

A. Frequently Asked Questions

This FAQ answers various questions related to the GNU Alternative Domain System, a fully decentralized and censorship-resistant replacement for DNS. These questions are based on actual questions, remarks or design suggestions we got from reviewers on a previous draft of a GADS paper. Some of the questions are explicitly or indirectly answered in this thesis; they are still presented here to make the answers easier to find.

What is the ".gads" top level domain?

The ".gads" top level domain (TLD) is the root of each user's personal name space. All GADS names have this TLD. The use of ".gads" as a TLD is really just a trick to combine the existing DNS hierarchy with the GADS graph as it is seen from an individual user's point of view.

Who runs the ".gads" top level domain? Each individual GADS user will run his own personal ".gads" top level domain (TLD).

To be more specific, for each user, his own zone is authoritative for the ".gads" TLD.

What is the ".zkey" top level domain?

The ".zkey" top level domain (TLD) provides a (reverse) mapping from globally unique hashes to public key. A record in the ".zkey" TLD would look like

`HEK9TEBUQ5AT5V3FLL0HHNDA12HGH6BIM04TN7RDVOQ5B7TIEU80.zkey`

and a lookup for this name would return a delegation to the zone of the respective public key. As hash codes are used for the names, the ".zkey" TLD is not memorable (but globally unique and secure).

The ".zkey" TLD is expected to be used internally by some protocols, such as e-mail, which may require a secure, globally unique address. However, we expect that ".zkey" names will be hidden from the user by mapping them to memorable names in the ".gads" TLD in most applications. Naturally, users that need secure and globally unique names could use ".zkey" anyway. This should be seen as similar to users typing in IP addresses instead of hostnames.

Who runs the ".zkey" TLD?

Nobody runs the ".zkey" TLD, as names in the ".zkey" TLD are hashes over public keys, no authority is needed. The ".zkey" TLD is essentially a very simple mapping mechanism where only public key 'PKEY' records are stored — each name is mapped to the corresponding PKEY value (which is again the name). Given this, no actual authority is needed: ".zkey" lookups never involve any network operation but rather only consist of a trivial conversion from ASCII to binary. The ".zkey" TLD is really

only a way for users to explicitly say that they want to access the zone of a given public key (hash).

What is a zone in GADS?

A zone in GADS (like DNS) is a portion of the name space that a single entity (usually a user) is responsible for. For example, GNU is responsible for the "gnu.org" zone and thus all names ending with ".gnu.org".

In GADS, you control personal ".gads" top-level domain, which is called your **master** zone. In other words, you are responsible for and have complete authority over all names in the zone ".gads". However, by using delegation you can declare that some other user is responsible for any subdomain in your ".gads" zone. For example, you may delegate "bob.gads" to your friend Bob. This is equivalent to delegation with NS records in DNS, except that GADS uses cryptographic keys to establish the delegation instead of delegating to an IP address.

Records in a GADS zone, by design, consist only of a single label. In DNS, subdomains (names with additional dots) may or may not belong to the same zone, and the delegation via NS records is "invisible" to the (normal) user. In GADS, you would typically add records for all of your hosts and services either under "+" (which is used to represent the empty name) or under a simple service name without dots ("."). Whenever multiple dots come into play in GADS, delegation to another zone is in play.

Example:
A domain name like www.bob.gads is resolved by first looking in the user's **master** zone for a PKEY (or NS) record under the name of "bob". Then, if "bob" is a PKEY record in that **master** zone, the "www" name will be resolved in Bob's **master** zone (which is most likely managed by a user called 'bob').

By default, each user in GADS is given control over three zones with different purposes; however, these purposes and functions are established by conventions and the user interface, not by the GADS protocol. The three zones are:

- The **master** zone, which should be used mostly for records that the owner wants to make available to other users.
- The Shorten zone, which can populated internally by your local GADS resolver to keep GADS names short.
- The Private zone, which is supposed to be used for private records which the user does not want to make public (like secretserver.private.gads).

Records in each user's **master** zone control under which names the Shorten and Private zones appear in the ".gads" TLD of the respective user.

What are the different zones in GADS for?

In GADS, each user controls three zones:

Master zone

This zone is your personal .gads Top-Level-Domain zone. Any record mapped directly into this zone (www.gads) is controlled directly by you.

Private zone

This zone can be used by users to add private records they don't ever want to make public. Very useful if you don't want to 'accidently' make some records public for privacy reasons (bank.private.gads)

Shorten zone

The Shorten zone is populated internally by the GADS resolver to keep GADS names short. The GADS proxy uses this zone to shorten long GADS names on HTML web pages.

What record types are supported by GADS?

With GADS you have most of the record types you know from legacy DNS plus some GADS specific record types.

- A, NS, CNAME, PTR, MX, TXT, AAAA, SRV
- PKEY: the hash over a public key
- PSEU: the desired pseudonym the user picked for his zone
- LEHO: a legacy hostname, used for SSL validation or virtual hosting. Usually found in combination with a corresponding A/AAAA records under the same name
- REV: record for zone revocation

Is there a graphical user interface?

Yes, for the configuration and management of the zones.

Do you really expect normal users to use the GADS zone editor?

We expect that the GADS zone editor will be used by roughly the same user base as equivalent DNS zone editors: administrators that run servers and services as well as advanced (or curious) users. Normal users should have no real need for the GADS zone editor, as they also do not host services.

For normal users, we expect that automatic name shortening will populate their shorten zone, and that they will import names from the shorten zone into their private or global zones using either (1) little helper programs integrated into their desktop or browser or (2) QR code readers which detect QR codes that were generated by service administrators and distributed using promotional materials. Those QR codes include a URI which when passed to 'gads-uri' will automatically update the user's global zone (unless the name is already taken). Thus, a QR reader that de-

tects a 'gads://'-URI can easily be used for adding delegations (PKEY records) to the global zone, which in addition to shortening is the only relevant operation for normal users.

Naturally, normal users may choose to use the GADS zone editor to remove names, but simple removal is obviously way simpler than expecting users to perform the full functionality of DNS/GADS record management that system administrators might require.

Where is the per-user GADS database kept?

The short answer is that the database is kept at the user's peer. Now, a user may run multiple peers, in which case the database could be kept at each peer (however, we don't have code for convenient replication). Similarly, multiple peers can share one instance of the database — the namestore service can be accessed from remote (via TCP). The actual data can be stored in a Postgres database, for which various replication options are again applicable. Ultimately, there are many options for how users can store (and secure) their GADS database. However, we expect that normal users will have their database on their local PC whereas expert users might have it on a router or "in the cloud".

What is the expected average size of a GADS namestore database?

Pretty small. Based on our user study where we looked at browser histories and the number of domains visited, we expect that GADS databases will only grow to a few tens of thousands of entries, small enough to fit even on mobile devices.

Is GADS resistant to attacks on DNS used by the US?

We believe so, as there is no entity that any government could force to change the mapping for a name except for each individual user (and then the changes would only apply to the names that this user is the authority for). So if everyone used GADS, the only practical attack of a government would be to force the operator of a server to change the GADS records for his server to point elsewhere. However, if the owner of the private key for a zone is unavailable for enforcement, the respective zone cannot be changed and any other zone delegating to this zone will achieve proper resolution.

Naturally, there are various attacks on the P2P network (a government might be able to filter all P2P traffic) and the DHT itself (i.e. Sybil and eclipse attacks); however, GADS uses a pretty robust DHT so we expect that the attacker would need to be quite resourceful to have a significant impact.

What is the difference between GADS and CoDoNS?

CoDoNS decentralizes the DNS database (using a DHT) but preserves the authority structure of DNS. With CoDoNS, IANA/ICANN are still in charge, and there are still registrars that determine who owns a name.

With GADS, we decentralize the database and also decentralize the responsibility for naming: each user runs his own personal root zone and is thus in complete control of the names he uses. GADS also has many additional features (to keep names short and enable migration) which don't even make sense in the context of CoDoNS.

What is the difference between GADS and SocialDNS?

Like GADS, SocialDNS allows each user to create DNS mappings. However, with SocialDNS the mappings are shared through the social network and subjected to ranking. As the social relationships evolve, names can thus change in surprising ways.

With GADS, names are primarily shared via delegation, and thus mappings will only change if the user responsible for the name (the authority) manually changes the record.

How does GADS compare to ODDNS?

ODDNS is primarily designed to bypass the DNS root zone and the TLD registries (such as those for ".com" and ".org"). Instead of using those, each user is expected to maintain a database of (second-level) domains (like "gnu.org") and the IP addresses of the respective name servers. Resolution will fail if the target name servers change IPs.

With GADS, delegation is not done using hard-coded IP addresses of DNS servers. Instead, GADS delegates to public keys and uses the P2P network to determine the current record information (which must be signed by the respective private key). Thus, resolution would not fail if the target name server is forced to change IP addresses.

Furthermore since GADS supports DNS delegations using so called NS records as well it is a simple matter of adding appropriate records to your zone to emulate ODDNSs behaviour.

Does GADS require real-world introduction (secure PKEY exchange) in the style of the PGP web of trust?

For security, it is well known that an initial trust path between the two parties must exist. However, for applications where this is not required, weaker mechanisms can be used. For example, we have implemented a first-come-first-served (FCFS) authority which allows arbitrary users to register arbitrary names. The key of this authority is included with every GADS installation. Thus, any name registered with FCFS is in fact global and requires no further introduction. However, the security of these names depends entirely on the trustworthiness of the FCFS authority. In contrast to DNS, there can be many such authorities in GADS (which users can call by any name they wish and change at any time) and obviously authorities can have policies other than FCFS.

As a result, naming authorities in GADS provide high trust agility (unlike DNS). And unlike trusted certificate authorities on X.509, it is much more inherently obvious which names a given GADS authority is allowed to assure: only its subdomains. Finally, users have complete freedom in selecting such authorities (and their names). We expect that users will use direct trust relationships for critical activities (such as banking and political resistance) and rely on semi-reliable third parties for entertainment and other uncritical functions.

How can a legitimate domain owner tell other people to not use his name in GADS?

Names have no owners in GADS, so there cannot be a "legitimate" domain owner. Any user can claim any name (as his preferred name or 'pseudonym') in his PSEU record. Similarly, all other users can choose to ignore this preference and use a name of their choice (or even assign no name) for this user.

An exception are the 'first-come first-served' authorities. Such authorities allow each user to register a pseudonym on a first-come first-served basis. Users can choose to delegate a subdomain (such as fcfs.gads) to such authorities. If the authority is not compromised and trustworthy (which it may not be!), those names would then "never" change. However, GADS can be used entirely without such authorities, they might just be convenient at times.

Why do you only allow one pseudonym (PSEU record) per user in GADS?

The basic idea behind the question is that one should allow users to suggest multiple pseudonyms (possibly with a ranking), and if one of the names is already taken (for shortening) GADS should use one of the alternative names.

While this would seem to decrease the chances of unresolvable naming conflicts, our rationale behind the design decision to only allow one pseudonym is that we want to maximize the incentive to users to pick a really good pseudonym. Essentially, if you have one and only choice — and if your choice is not good, it will be completely ignored — we hope that users will think harder about this choice. The analogous situation is that making users create one good password is better than getting a dozen bad passwords. However, we freely admit that we have no hard data to confirm that this design decision is correct.

What can I do if name shortening is not desired for a particular zone (such as ai.mit.gads)?

If the PSEU record is left out, GADS will not apply shortening. This can be done in the GADS zone editor by leaving the pseudonym blank. If this is done for the 'ai' zone, then a delegation from the 'mit' zone to the 'ai' zone will never be shortened to 'ai.short.gads'. However, other users can still manually give the 'ai' zone any name they wish (for example, 'ki-mit.gads') — it just won't happen automatically.

Did you consider the privacy implications of making your personal GADS zone visible?

Each record in GADS has a flag "private". Records are shared with other users (via DHT or zone transfers) only if this flag is not set. Thus, users have full control over what information about their zones is made public.

In particular, records that GADS automatically adds (i.e. via name shortening) are always marked 'private' by default. Otherwise, other users might indeed be able to obtain sensitive private information about one's online behavior.

Note that while we encourage users to put their 'private' records into the special "private zone", this is not required and records with the "private" flag in the global or shorten zone also would not be available to other users.

In GADS, does shortening to pseudonyms picked by other users facilitate phishing attacks?

To a limited degree, yes. I can pick my pseudonym to be "bank" and then if then someone else's peer learns about my identity his client would refer to me as "bank", even though I'm unlikely to be the bank of the other user. However, GADS mitigates this problem by placing all shortened records into the shorten zone, so the name will occur as bank.shorten.gads, not bank.gads. This hopefully will give most users a strong visual hint. If you believe that this is insufficient, shortening can be disabled.

Are "Legacy Host" (LEHO) records not going to be obsolete with IPv6?

The question presumes that (a) virtual hosting is only necessary because of IPv4 address scarcity, and (b) that LEHOs are only useful in the context of virtual hosting. However, LEHOs are also useful to help with X.509 certificate validation (as they specify for which legacy hostname the certificate should be valid). Also, even with IPv6 fully deployed and "infinite" IP addresses being available, we're not sure that virtual hosting would disappear. Finally, we don't want to have to wait for IPv6 to become commonplace, GADS should work with today's networks.

Why does GADS not use a trust metric or consensus to determine globally unique names?

Trust metrics have the fundamental problem that they have thresholds. As trust relationships evolve, mappings would change their meaning as they cross each others thresholds. We decided that the resulting unpredictability of the resolution process was not acceptable. Furthermore, trust and consensus might be easy to manipulate by adversaries.

How do you handle compromised zone keys in GADS?

The owner of a private key can create a REVocation record in his zone file (under the name "+"). Once such a record exists, all peers will consider all records in this zone

to be invalid. All names that involve delegation (PKEY) via a revoked zone will then fail to resolve. Peers always automatically check for the existence of REV records when resolving names.

Could the signing algorithm of GADS be upgraded in the future?

Yes. Naturally, deployed GADS implementations would have to be updated to support the new signature scheme. The new scheme could then be run in parallel with the existing system by using a new record type (PKEY2) to indicate the use of a different cipher system.

Does GADS require a zone's signing keys to be online?

Right now, the simple answer is yes. The reason is that if a relative expiration time is given for a records (i.e. 1 week from now), each time a request for that name is received, a signature is created with an absolute expiration time of 1 week into the future. The simplest implementation for this uses a signing key that is directly available to the resolver.

In the future, two variations of this scheme are conceivable. First, we could have two keys, one that we use for signing online for a limited period of time and a second one that persists for a long time (or forever) which is only used to periodically sign the online signing key and otherwise kept offline.

Another variation would be to pre-generate signatures for all records periodically (i.e. for the next week) and then take the signing key offline again.

We have currently implemented neither scheme as (1) GADS signing keys are likely not all that valuable — most normal users can easily create a new one with little loss, key loss is likely only critical for entities such as banks where there are financial risks and where there are significant costs to securely provide their users with their new key; (2) we would like the first implementation to be usable; complex key management operations with online and offline keys are not going to help here; (3) given that it is not clear to what extend this is needed, we feel that especially the first design and implementation should not be burdened by possibly unnecessary complexity. To introduce signing keys, all we would probably need is a new block type and an additional record type, so forward-compatibility is not expected to be a major roadblock should separate keys be required in the future.

How can a GADS zone maintain several name servers, e.g. for load balancing?

We don't expect this to be necessary, as GADS records are stored (and replicated) in the R^5N DHT. Thus the authority will typically not be contacted whenever clients perform a lookup. Even if the authority goes (temporarily) off-line, the DHT will cache the records for some time. However, should having multiple servers for a zone be considered truly necessary, the owner of the zone can simply run multiple peers (and share the zone's key and database among them).

Why are you intercepting DNS queries instead of running a DNS resolver?

Our system allows running a DNS resolver instead of using the interception approach. However, in order to run a personal zone, we would need to run a DNS

server for each user, not just for each host (which is at least a theoretical problem on multi-user systems, as most operating systems only allow one DNS server to be configured per host). Note that the firewall-based DNS interception suffers from the same problem.

Does translating names in GADS break browser's same-origin policy?

The usual mapping of names in GADS is unproblematic as the browser either does not really see it (with the GADS proxy) or does it itself (in which case policy code would just have to be adjusted). However, there are issues in particular cases which the GADS proxy needs to handle.

The same-origin policy (SOP) of a browser tries to ensure that information (Cookies and JavaScript in particular) obtained from one site (i.e. "gnu.org") does not interfere with information from another site (i.e. "fsf.org"). At the surface, this does not fundamentally change with GADS; the browser would just have to check that "alice.bob.gads" does not interfere with 'dave.bob.gads' or "bob.gads". SOP already has rules to deal with special cases to deal with the fact that entities below "co.uk" are not related, so SOP for ".gads" can simply assume that only names under ".gads" are related if they match exactly.

Existing websites can create two additional problems. First, some websites include resources (such as JavaScript or CSS files) using absolute host names in the HTML. Example: "alice.gads" might serve a website which includes resources from `http://alice.com/resources`. This would be fine if the browser actually accessed "alice.com", but as the browser now sees "alice.gads", falling back to DNS names for embedded documents can create problems. The GADS proxy solves this problem by detecting that "alice.com" is the LEHO value of 'alice.gads' and then transforms the link. A better solution would have been for Alice to include a relative link to "/resources", which would have worked with DNS and GADS.

A second problem is that sometimes websites set cookies for entire subdomains. For example, "www.gnu.org" might set a cookie for "*.gnu.org". With GADS and the SOP described above, this would not be allowed. However, browsers that fully support GADS would see that "www.gnu.gads" and "lists.gnu.gads" are both names under the same GADS authority (same public key) and can thus decide that they are the same origin. This is in fact a cleaner way to determine that two names belong to the same authority than the heuristics used with the current DNS system.

Finally, if GADS delegates to DNS via an NS record, the browser can and should probably assume that the resulting subdomain is a different authority. This would only cause problems if GADS records perform NS delegation to DNS TLDs or even IANA (for example, to achieve something like gnu.org.iana.gads being an alias for gnu.org); here, clearly not all subdomains under 'iana.gads' are the same origin. But this is more of a theoretical problem when it comes to integrating with legacy DNS.

Will GADS work with cookies?

GADS should work fine with cookies in most cases. The GADS proxy translates cookies set by the browser for "gnu.org" to the domain name the browser expects (i.e. gnu.gads). Similarly, if the webserver believes it is "alice.gads" the GADS proxy can translate cookies to "alice.bob.gads".

The problematic case is webservers setting cookies for entire subdomains. For example "www.gnu.org" setting a cookie for "*.gnu.org". Here, the GADS proxy needs to essentially check if all of the domains the cookie is to be set for fall under the same origin. In GADS, the same origin is easily determined as all records (that are not NS or PKEY records) signed by the same public key can be assumed to belong to the same authority. For further details, see the FAQ entry on GADS and the same origin policy.

How will existing network protocols cope with a transition from DNS to GADS?

This depends of course largely on the protocol. Our documentation and implementation efforts have largely focused on HTTP/HTTPS as this is the dominant protocol in use and here the devil is sometimes in the details. Some other protocols — such as most P2P protocols — do not really use DNS and would thus not be affected by a DNS-GADS transition.

Protocols like SMTP will require some work in the software stack (which can again often be done using proxies) to translate GADS names in the appropriate places. We have not yet encountered a protocol that absolutely cannot be migrated, but if you have a specific concern we would like to hear from you.

B. Command-Line Tool Reference

B.1. gnunet-namestore (1)

B.1.1. Name

gnunet-namestore - manipulate GNS zones

B.1.2. Synopsis

gnunet-namestore [*options*]

B.1.3. Description

gnunet-namestore can be used to create and manipulate a GNS zone.

B.1.4. Options

-a, --add: Desired operation is adding a record

-c FILENAME, --config=FILENAME Use the configuration file FILENAME.

-d, --delete: Desired operation is deleting a record

-D, --display: Desired operation is listing of matching records

-e TIME, --expiration=TIME: Specifies expiration time of record to add; format is relative time, i.e "1 h" or "7 d 30 m". Supported units are "ms", "s", "min" or "minutes", "h" (hours), "d" (days) and "a" (years).

-h, --help: Print short help on options.

-L LOGLEVEL, --loglevel=LOGLEVEL: Use LOGLEVEL for logging. Valid values are DEBUG, INFO, WARNING and ERROR

-n NAME, --name=NAME: Name of the record to add/delete/display

-t TYPE, --type=TYPE: Type of the record to add/delete/display (i.e. "A", "AAAA", "NS", "PKEY", "MX" etc.)

-u URI, --uri=URI: Add PKEY record from gnunet://gns/-URI to our zone; the record type is always PKEY, if no expiration is given FOREVER is used

-v, --version: Print GNUnet version number.

-V VALUE, --value=VALUE: Value to store or remove from the GNS zone. Specific format depends on the record type. A records expect a dotted decimal IPv4 address, AAAA records an IPv6 address, PKEY a public key in GNUnet's printable format, and CNAME and NS records should be a domain name.

-z FILENAME, --zonekey=FILENAME: Specifies the filename with the private key for the zone (mandatory option)

B.1.5. Bugs

Report bugs by using Mantis <https://gnunet.org/bugs/> or by sending electronic mail to <gnunet-developers@gnu.org>

B.1.6. See Also

gnunet-gns(1)

B.2. gnunet-gns (1)

B.2.1. Name

gnunet-gns - Access to GNUnet Name Service

B.2.2. Synopsis

gnunet-gns [*options*]

B.2.3. Description

gnunet-gns can be used to lookup and process GNUnet Name Service names.

B.2.4. Options

-a NAME, --authority=NAME Get the authority of a particular name. For example the authority for "www.fcfs.gads" is "fcfs.gads".

-c FILENAME, --config=FILENAME Use the configuration file FILENAME.

-r, --raw No unneeded output. This is a quiet mode where only important information is displayed. For example a lookup for an IP address will only yield the IP address, no descriptive text.

-s NAME, --shorten NAME Shorten GNUnet Name Service Name. The service will try to shorten the delegation chain of the name if a "closer" authority chain exists relative to your local root zone.

-t RRTYPE, --type=RRTYPE Resource Record Type (RRTYPE) to look for. Supported RRTYPE's are: A, AAAA, CNAME, NS, PKEY, PSEU, TLSA, SRV, SOA, MX, LEHO, VPN, REV, PTR, TXT Defaults to "A".

-h, --help Print short help on options.

-L LOGLEVEL, --loglevel=LOGLEVEL Use LOGLEVEL for logging. Valid values are DEBUG, INFO, WARNING and ERROR.

-u NAME, --lookup=NAME Name to lookup. Resolve the specified name using the GNUnet Name System.

-v, --version Print GNUnet version number.

B.2.5. Bugs

Report bugs by using Mantis <https://gnunet.org/bugs/> or by sending electronic mail to <gnunet-developers@gnu.org>

B.2.6. See Also

gnunet-namestore(1)

C. GNUnet Name System API

C.1. Function Documentation

C.1.1. void GNUNET_GNS_cancel_get_auth_request (struct GNUNET_GNS_GetAuthRequest * *gar*)

Cancel pending get auth request

Parameters

gar	the lookup request to cancel

C.1.2. void GNUNET_GNS_cancel_lookup_request (struct GNUNET_GNS_LookupRequest * *lr*)

Cancel pending lookup request

Parameters

lr	the lookup request to cancel

C.1.3. void GNUNET_GNS_cancel_shorten_request (struct GNUNET_GNS_ShortenRequest * *sr*)

Cancel pending shorten request

Parameters

sr	the lookup request to cancel

C.1.4. struct GNUNET_GNS_Handle* GNUNET_GNS_connect (const struct GNUNET_CONFIGURATION_Handle * *cfg*)

Initialize the connection with the GNS service.

Parameters

cfg	configuration to use

Returns

handle to the GNS service, or NULL on error

C.1.5. void GNUNET_GNS_disconnect (struct GNUNET_GNS_Handle * *handle*)

Shutdown connection with the GNS service.

Parameters

handle	connection to shut down

C.1.6. struct GNUNET_GNS_GetAuthRequest* GNUNET_GNS_get_authority (struct GNUNET_GNS_Handle * *handle*, const char * *name*, GNUNET_GNS_GetAuthResultProcessor *proc*, void * *proc_cls*)

Perform an authority lookup for a given name.

Parameters

handle	handle to the GNS service
name	the name to look up authority for
proc	function to call on result
proc_cls	closure for processor

Returns

handle to the operation

C.1.7. struct GNUNET_GNS_LookupRequest* GNUNET_GNS_lookup (struct GNUNET_GNS_Handle * *handle*, const char * *name*, enum GNUNET_GNS_RecordType *type*, int *only_cached*, struct GNUNET_CRYPTO_RsaPrivateKey * *shorten_key*, GNUNET_GNS_LookupResultProcessor *proc*, void * *proc_cls*)

Perform an asynchronous lookup operation on the GNS in the default zone.

Parameters

handle	handle to the GNS service
name	the name to look up
type	the GNUNET_GNS_RecordType to look for

only_cached	GNUNET_NO to only check locally not DHT for performance
shorten_key	the private key of the shorten zone (can be NULL)
proc	function to call on result
proc_cls	closure for processor

Returns

handle to the queued request

C.1.8. struct GNUNET_GNS_LookupRequest* GNUNET_GNS_lookup_zone (struct GNUNET_GNS_Handle * *handle*, const char * *name*, struct GNUNET_CRYPTO_ShortHashCode * *zone*, enum GNUNET_GNS_RecordType *type*, int *only_cached*, struct GNUNET_CRYPTO_RsaPrivateKey * *shorten_key*, GNUNET_GNS_LookupResultProcessor *proc*, void * *proc_cls*)

Perform an asynchronous lookup operation on the GNS in the zone specified by 'zone'.

Parameters

handle	handle to the GNS service
name	the name to look up
zone	the zone to start the resolution in
type	the GNUNET_GNS_RecordType to look for
only_cached	GNUNET_YES to only check locally not DHT for performance
shorten_key	the private key of the shorten zone (can be NULL)
proc	function to call on result
proc_cls	closure for processor

Returns

handle to the queued request

C.1.9. struct GNUNET_GNS_ShortenRequest* GNUNET_GNS_shorten (struct GNUNET_GNS_Handle * *handle*, const char * *name*, struct GNUNET_CRYPTO_ShortHashCode * *private_zone*, struct GNUNET_CRYPTO_ShortHashCode * *shorten_zone*, GNUNET_GNS_ShortenResultProcessor *proc*, void * *proc_cls*)

Perform a name shortening operation on the GNS.

Parameters

handle	handle to the GNS service
name	the name to look up

private_zone	the public zone of the private zone
shorten_zone	the public zone of the shorten zone
proc	function to call on result
proc_cls	closure for processor

Returns

handle to the operation

C.1.10. struct GNUNET_GNS_ShortenRequest∗ GNUNET_GNS_shorten_zone (struct GNUNET_GNS_Handle ∗ *handle*, const char ∗ *name*, struct GNUNET_CRYPTO_ShortHashCode ∗ *private_zone*, struct GNUNET_CRYPTO_ShortHashCode ∗ *shorten_zone*, struct GNUNET_CRYPTO_ShortHashCode ∗ *zone*, GNUNET_GNS_ShortenResultProcessor *proc*, void ∗ *proc_cls*)

Perform a name shortening operation on the GNS.

Parameters

handle	handle to the GNS service
name	the name to look up
private_zone	the public zone of the private zone
shorten_zone	the public zone of the shorten zone
zone	the zone to start the resolution in
proc	function to call on result
proc_cls	closure for processor

Returns

handle to the operation

D. GADS Record Types and Flags

D.1. Record Types

Record Name	Record Number
ANY	0
A	1
NS	2
CNAME	5
SOA	6
PTR	12
MX	15
TXT	16
AAAA	28
SRV	33
TLSA	52
PKEY	65536
PSEU	65537
LEHO	65538
VPN	65539
REV	65540

D.2. Record Flags

Record Flag	Flag Number	Usage
NONE	0	No special options
AUTHORITY	1	This peer is the authority for this record; it must thus not be deleted (other records can be deleted if we run out of space).
PRIVATE	2	This is a private record of this peer and it should thus not be handed out to other peers.
PENDING	4	This record was added by the system and is pending user confimation
RELATIVE_EXPIRATION	8	This expiration time of the record is a relative time (not an absolute time).
SHADOW_RECORD	16	This record should not be used unless all (other) records with an absolute expiration time have expired.

E. Browsing Survey

E.1. Scripts

This is a listing of the scripts we used in our user survey. The scripts crawl the local history databases of Firefox and Chrome browsers respectively.

E.1.1. Chromium and Chrome

```
1  #!/bin/sh
2  # Please run this shell script using "History" as the argument.
3  # You need to have 'sqlite3' installed.  The 'History' file
4  # is usually in
5  # Chromium on linux: ~/.config/chromium/Default/
6  # Google Chrome: ~/.config/google-chrome/Default/
7  #
8  LINKS_FOLLOWED=`echo "select count(*) from urls where typed_count=0;"
9                  | sqlite3 "$1"`
10 LINKS_TYPED=`echo "select count(*) from urls where typed_count>0;"
11               | sqlite3 "$1"`
12 DOMAINS_FOLLOWED_UNIQUE=`echo "select url from  urls where typed_count=0;"
13                           | sqlite3 "$1"
14                           | sed -e "s/https:\/\///" -e
15                           "s/http:\/\///" -e "s/\/.*//" -e "s/:.*//"
16                           | grep -v \\\hline\[ | sort | uniq | wc -l`
17 DOMAINS_TYPED_UNIQUE=`echo "select url from  urls where typed_count>0;"
18                           | sqlite3 "$1"
19                           | sed -e "s/https:\/\///" -e
20                           "s/http:\/\///" -e "s/\/.*//" -e "s/:.*//"
21                           | grep -v \\\hline\[ | sort | uniq | wc -l`
22 echo "You followed $LINKS_FOLLOWED links and typed in $LINKS_TYPED"
23 echo "You followed $LINKS_FOLLOWED links to $DOMAINS_FOLLOWED_UNIQUE
24       unique domains"
25 echo "You typed in $LINKS_TYPED links to
26       $DOMAINS_TYPED_UNIQUE unique domains"
27 echo " "
28 echo "Please stand by, this will take a moment..."
29 DOMAINS_TYPED=`echo "select id,url from urls where typed_count>0;"
30                 | sqlite3 "$1"
31                 | sed -e "s/https:\/\///" -e "s/http:\/\///" -e
32                 "s/\/.*//" -e "s/:.*//"
33                 | grep -v \\\hline\[ | sort | uniq`
34 PRIOR=0
35 CNT=0
36 SEENTMP=`mktemp /tmp/seenXXXXXX`
37 for n in $DOMAINS_TYPED
```

```
38 do
39  ID=`echo $n | sed -e "s/|.*//"`
40  DOM=`echo $n | sed -e "s/.*|//"`
41  if ! grep "^$DOM\$" $SEENTMP > /dev/null
42  then
43    CNT=`expr $CNT + 1`
44    echo "$DOM" >> $SEENTMP
45    DOMAINS_PRIOR=`echo "select url from urls where
46                      typed_count=0 AND id < $ID;"
47                      | sqlite3 "$1"
48                      | sed -e "s/https:\/\///" -e
49                      "s/http:\/\///" -e "s/\/.*//" -e
50                      "s/:.*//" | sort | grep $DOM
51                      | head -n 1 | wc -l`
52    PRIOR=`expr $PRIOR + $DOMAINS_PRIOR`
53  fi
54 done
55 # rm $SEENTMP
56 echo " "
57 echo "Of $CNT domains typed in, $PRIOR had been
58       visited previously (by ID) via links"
59 echo "Summary: $LINKS_FOLLOWED $LINKS_TYPED
60       $DOMAINS_FOLLOWED_UNIQUE $DOMAINS_TYPED_UNIQUE $PRIOR"
61 echo " "
62 echo "Please e-mail the output to gns-data@gnunet.org"
```

E.1.2. Firefox

```
1
2  #!/bin/sh
3  # Please run this shell script using "places.sqlite" as the argument.
4  # You need to have 'sqlite3' installed. The 'places.sqlite' file
5  # is usually in ~/.mozilla/firefox/RANDOMDIRNAME/places.sqlite
6  #
7  LINKS_FOLLOWED=`echo "select count(*) from moz_places where typed=0;"
8                  | sqlite3 "$1"`
9  LINKS_TYPED=`echo "select count(*) from moz_places where typed=1;"
10               | sqlite3 "$1"`
11 DOMAINS_FOLLOWED_UNIQUE=`echo "select url from moz_places where typed=0;"
12                          | sqlite3 "$1"
13                          | sed -e "s/https:\/\///" -e
14                          "s/http:\/\///" -e "s/\/.*//" -e
15                          "s/:.*//" | grep -v \\\hline\[
16                          | sort | uniq | wc -l`
17 DOMAINS_TYPED_UNIQUE=`echo "select url from moz_places where typed=1;"
18                       | sqlite3 "$1"
19                       | sed -e "s/https:\/\///" -e "s/http:\/\///"
20                       -e "s/\/.*//" -e "s/:.*//"
21                       | grep -v \\\hline\[ | sort
22                       | uniq | wc -l`
23 echo "You followed $LINKS_FOLLOWED links and typed in $LINKS_TYPED"
24 echo "You followed $LINKS_FOLLOWED links to
25       $DOMAINS_FOLLOWED_UNIQUE unique domains"
26 echo "You typed in $LINKS_TYPED links to
```

```
27        $DOMAINS_TYPED_UNIQUE unique domains"
28 echo " "
29 echo "Please stand by, this will take a moment..."
30 DOMAINS_TYPED=`echo "select id,url from moz_places where typed=1;"
31               | sqlite3 "$1"
32               | sed -e "s/https:\/\///" -e "s/http:\/\///"
33               -e "s/\/.*//" -e "s/:.*//"
34               | grep -v \\\hline\[ | sort | uniq`
35 PRIOR=0
36 CNT=0
37 SEENTMP=`mktemp /tmp/seenXXXXXX`
38 for n in $DOMAINS_TYPED
39 do
40  ID=`echo $n | sed -e "s/|.*//"`
41  DOM=`echo $n | sed -e "s/.*|//"`
42  if ! grep "^$DOM\$" $SEENTMP > /dev/null
43  then
44    CNT=`expr $CNT + 1`
45    echo "$DOM" >> $SEENTMP
46    DOMAINS_PRIOR=`echo "select url from moz_places
47                        where typed=0 AND id < $ID;"
48                  | sqlite3 "$1" | sed -e "s/https:\/\///"
49                  -e "s/http:\/\///" -e "s/\/.*//" -e "s/:.*//"
50                  | sort | grep $DOM | head -n 1 | wc -l`
51    PRIOR=`expr $PRIOR + $DOMAINS_PRIOR`
52  fi
53 done
54 # rm $SEENTMP
55 echo " "
56 echo "Of $CNT domains typed in, $PRIOR had
57       been visited previously (by ID) via links"
58 echo "Summary: $LINKS_FOLLOWED $LINKS_TYPED
59                $DOMAINS_FOLLOWED_UNIQUE
60                $DOMAINS_TYPED_UNIQUE $PRIOR"
61 echo " "
62 echo "Please e-mail the output to gns-data@gnunet.org"
```

E.2. User Data

This section consists of the data acquired in our user survey using the scripts in Appendix E.1.

User	Followed	Typed	UD	TUD	PV TUD	FT	Percentage
0	57651	1786	4313	464	190	274	6.3528866218
1	11457	591	2353	198	74	124	5.2698682533
2	94068	5406	6025	944	518	426	7.0705394191
3	14114	363	1747	210	82	128	7.3268460218
4	6639	134	997	87	33	54	5.4162487462
5	18728	230	1129	190	44	146	12.9317980514
6	13178	199	1856	139	66	73	3.9331896552
7	22407	208	3513	126	65	61	1.7364076288
8	32898	336	4758	210	123	87	1.8284993695
9	5726	206	908	94	42	52	5.7268722467
10	2665	103	457	86	12	74	16.1925601751
11	4485	157	709	138	33	105	14.8095909732
12	17799	483	1714	197	60	137	7.9929988331
13	5475	89	643	56	22	34	5.2877138414
14	5608	159	840	149	40	109	12.9761904762
15	14854	208	1244	155	54	101	8.1189710611
16	16566	403	1809	219	81	138	7.6285240464
17	6540	181	782	131	52	79	10.1023017903
18	35886	897	4747	551	291	260	5.477143459
19	39415	585	4158	369	171	198	4.7619047619
20	61374	7999	7439	1947	649	1298	17.4485817986
21	8173	337	713	172	51	121	16.9705469846
22	44826	1540	4246	688	317	371	8.7376354216
23	13696	723	1836	309	130	179	9.7494553377
24	250	8	133	8	0	8	6.015037594
25	1210	51	576	38	16	22	3.8194444444
26	6259	84	650	65	22	43	6.6153846154
27	296	10	92	9	0	9	9.7826086957
28	1691	57	235	51	12	39	16.5957446809
29	93	0	73	0	0	0	0
30	27841	712	2100	168	87	81	3.8571428571
31	11042	145	1058	123	44	79	7.4669187146
32	20692	207	623	143	34	109	17.4959871589
33	16499	286	322	85	26	59	18.3229813665
34	2859	143	578	91	36	55	9.5155709343
35	2469	125	508	88	36	52	10.2362204724
36	698	18	74	16	1	15	20.2702702703
37	22288	684	2701	346	161	185	6.8493150685
38	627	9	83	5	3	2	2.4096385542
39	678	13	71	8	1	7	9.8591549296
40	4723	114	612	50	19	31	5.0653594771
41	21133	657	1884	407	126	281	14.91507431
42	5045	310	1030	233	75	158	15.3398058252
43	24791	625	2785	487	220	267	9.5870736086

44	13936	277	1625	175	71	104	6.4
45	4886	224	774	80	39	41	5.2971576227
46	8712	608	1259	329	100	229	18.1890389198
47	19282	162	1782	135	58	77	4.3209876543
48	53457	753	4733	501	219	282	5.958166068
49	17838	459	1720	177	63	114	6.6279069767
50	91471	3043	8701	1492	551	941	10.8148488679
51	2142	196	302	147	22	125	41.3907284768
52	6967	235	897	130	38	92	10.2564102564
53	8083	494	1210	263	102	161	13.305785124
54	9213	160	1287	129	50	79	6.1383061383
55	5018	386	808	89	34	55	6.8069306931
56	13500	1361	2241	606	201	405	18.0722891566
57	51733	754	5195	483	204	279	5.3705486044
58	235	73	38	29	7	22	57.8947368421
59	305	95	239	85	9	76	31.7991631799
Avg	1027172	36861	107935	15100	5887	9213	8.5356927781

Followed	Number of links followed
Typed	Number of domains typed
UD	Unique Domains
TUD	Typed Unique Domains
PV TUD	Previously Typed Unique Domains
FT	Freshly Typed

Bibliography

[1] *Information technology: automatic identification and data capture techniques, QR code 2005 bar code symbology specification*. BSI Group, London, 2009.

[2] R. Arends, R. Austein, M. Larson, D. Massey, and S. Rose. DNS Security Introduction and Requirements. *IETF RFC 4033*, Mar. 2005.

[3] R. Barnes. Rfc 6394: Use cases and requirements for dns-based authentication of named entities (dane). `https://datatracker.ietf.org/wg/dane/charter/`, October 2011.

[4] Burton H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 13:422–426, 1970.

[5] Guihai Chen, Tongqing Qiu, and Fan Wu. Insight into redundancy schemes in dhts. *Journal of Supercomputing*, 43:183–198, 02/2008 2008.

[6] Thibault Cholez, Isabelle Chrisment, and Olivier Festor. Evaluation of sybil attacks protection schemes in kad. In *AIMS'09 - Proceedings of the 3rd International Conference on Autonomous Infrastructure, Management and Security: Scalability of Networks and Services*, volume 5637 of *Lecture Notes in Computer Science*, page 70–82, Enschede, The Netherlands, 06/2009 2009. Springer-Verlag, Springer-Verlag.

[7] F. R. K. Chung. *Spectral Graph Theory*. American Mathematical Society, 1997.

[8] Frank Dabek, Jinyang Li, Emil Sit, James Robertson, Frans M. Kaashoek, and Robert Morris. Designing a dht for low latency and high throughput. In *NSDI'04 - Proceedings of the 1st conference on Symposium on Networked Systems Design and Implementation*, page 7–7, San Francisco, CA, USA, 03/2004 2004. USENIX Association, USENIX Association.

[9] George Danezis, Chris Lesniewski-laas, Frans M. Kaashoek, and Ross Anderson. Sybil-resistant dht routing. In *ESORICS*, page 305–318. Springer, Springer, 2005.

[10] Daniel J. Bernstein and Tanja Lange (editors). ebacs: Ecrypt benchmarking of cryptographic systems. `http://bench.cr.yp.to/`, accessed 7 March 2013.

[11] Roger Dingledine, Nick Mathewson, and Paul Syverson. Tor: The second-generation onion router. In *Proceedings of the 13th USENIX Security Symposium*, August 2004.

[12] John R. Douceur. The sybil attack. In Peter Druschel, M. Frans Kaashoek, and Antony I. T. Rowstron, editors, *IPTPS*, volume 2429 of *Lecture Notes in Computer Science*, pages 251–260. Springer, 2002.

[13] Peter Eckersley. Souverign key cryptography for internet domains. `https://git.eff.org/?p=sovereign-keys.git`, December 2011.

[14] European Parliament. Resolution on the EU-US Summit of 28 November 2011, November 2011. P7-RC-2011-0577.

[15] Nathan Evans and Christian Grothoff. R5n: Randomized recursive routing for restricted-route networks. In *5th International Conference on Network and System Security*, pages 316–321, Milan, Italy, 2011. IEEE.

[16] P. Faltstrom, P. Hoffman, and A. Costello. Rfc 3490: Internationalizing domain names in applications (idna). Technical report, Network Working Group, March 2003.

[17] Amos Fiat, Jared Saia, and Maxwell Young. Making chord robust to byzantine attacks. In *Proc. of the European Symposium on Algorithms (ESA*, pages 803–814. Springer, Springer, 2005.

[18] R. Fielding. Rfc 1808: Relative uniform resource locators. Technical report, Network Working Group, June 1995.

[19] Free Software Foundation. The gnu c library - system databases and name service switch. `http://www.gnu.org/software/libc/manual/html_node/Name-Service-Switch.html`.

[20] Evgeniy Gabrilovich and Alex Gontmakher. The homograph attack. *Communications of the ACM*, 45(2), February 2002.

[21] Michael T. Goodrich, Michael J. Nelson, and Jonathan Z. Sun. The rainbow skip graph: a fault-tolerant constant-degree distributed data structure. In *SODA '06: Proceedings of the seventeenth annual ACM-SIAM symposium on Discrete algorithm*, page 384–393, New York, NY, USA, 2006. ACM, ACM.

[22] Stefan Götz, Simon Rieche, and Klaus Wehrle. *Selected DHT Algorithms*, volume 3485 of *Lecture Notes in Computer Science*, chapter 8, pages 95–117. Springer, 2005.

[23] Reuben Grinberg. Bitcoin: An innovative alternative digital currency. *Hastings Science & Technology Law Journal*, 4:159–208, December 2011.

[24] Martin Holland. Daenischer Polizist sperrt versehentlich 8000 Websites. `http://heise.de/-1447571`, March 2012.

[25] Ralph Holz, Lothar Braun, Nils Kammenhuber, and Georg Carle. The SSL landscape – a thorough analysis of the X.509 PKI using active and passive measurements. In *Proc. 11th Annual Internet Measurement Conference (IMC'11), Berlin, Germany*. ACM, Sheridan, Nov 2011.

[26] Iso 3166: Country codes. Technical report, International Organization for Standardization.

[27] Edwin Jacobs. Bitcoin: A bit too far? *Journal of Internet Banking and Commerce*, 16(2), August 2011.

[28] S. Josefsson. Rfc 4648: The base16, base32, and base64 data encodings. Technical report, Network Working Group, October 2006.

[29] Frans M. Kaashoek and David Karger. *Koorde: A Simple degree-optimal distributed hash table*, volume 2735/2003 of *Lecture Notes in Computer Science*, pages 98–107. Springer, Berlin / Heidelberg, 2003.

[30] Maxim Krasnyansky. Universal tun/tap device driver. `http://www.kernel.org/doc/Documentation/networking/tuntap.txt`.

[31] Susan Landau. Security, wiretapping, and the internet. *IEEE Security & Privacy*, pages 26–33, 2005.

[32] Ben Laurie and Adam Langley. Certificate transparency. `http://www.certificate-transparency.org/`, 2012. [last retrieved in April 2012].

[33] Y. Lee, M. Leech, and D. Koblas. Rfc 1928: Socks protocol version 5. Technical report, Network Working Group, March 1996.

[34] Chris Lesniewski-Laas and M. Frans Kaashoek. Whanau: a sybil-proof distributed hash table. In *Proceedings of the 7th USENIX conference on Networked systems design and implementation*, NSDI'10, pages 111–126, Berkeley, CA, USA, 2010. USENIX Association.

[35] Thomas Locher, David Mysicka, Stefan Schmid, and Roger Wattenhofer. Poisoning the kad network. In *ICDCN'10 - Proceedings of the 11th International Conference on Distributed Computing and Networking*, ICDCN'10, page 195–206, Kolkata, India, January 2010. Springer-Verlag, Springer-Verlag.

[36] M. Marlinspike. Null prefix attacks against ssl/tls certificates. http://www.thoughtcrime.org/papers/null-prefix-attacks.pdf, 2009.

[37] Moxie Marlinspike. Ssl and the future of authenticity. `http://blog.thoughtcrime.org/ssl-and-the-future-of-authenticity`, Arpil 2011.

[38] Prateek Mittal, Matthew Caesar, and Nikita Borisov. X-vine: Secure and pseudonymous routing using social networks. *CoRR*, abs/1109.0971, 2011.

[39] P. Mockapetris. Domain Names - Implementation and Specification. *IETF RFC 1035*, Nov. 1987.

[40] Paul Mockapetris. Rfc 1034: Domain names - concepts and facilities. Technical report, Network Working Group, November 1987.

[41] Paul Mockapetris. Rfc 1035: Domain names - implementation and specification. Technical report, Network Working Group, November 1987.

[42] Mozilla Security Blog. DigiNotar removal follow up. `https://blog.mozilla.com/security/2011/09/02/diginotar-removal-follow-up/` [online; last retrieved in September 2011], 2011.

[43] Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system. `http://bitcoin.org/bitcoin.pdf`, 2008.

[44] J. Postel. Rfc 1591: Domain name system structure and delegation. Technical report, Network Working Group, March 1994.

[45] Venugopalan Ramasubramanian and Emin Gun Sirer. The design and implementation of a next generation name service for the internet. In *Proceedings of SIGCOMM*, Portland, Oregon, August 2004.

[46] Ronald L. Rivest and Butler Lampson. Sdsi – a simple distributed security infrastructure. `http://groups.csail.mit.edu/cis/sdsi.html`, 1996.

[47] Rodrigo Rodrigues and Barbara Liskov. High availability in dhts: Erasure coding vs. replication. In *IPTPS'05 - Proceedings of the 4th International Workshop in Peer-to-Peer Systems*, volume 3640 of *Lecture Notes in Computer Science*, Ithaca, New York, 02/2005 2005. Springer, Springer.

[48] Alex Fink Sai. Mnemonic .onion urls. `https://gitweb.torproject.org/torspec.git/blob/HEAD:/proposals/194-mnemonic-urls.txt`, February 2012.

[49] C. Soghoian and S. Stamm. Certified lies: Detecting and defeating government interception attacks against SSL. In *Proc. 15th. Int. Conf. Financial Cryptography and Data Security*, Mar 2011.

[50] Richard Stallman. Why software should not have owners. `http://www.gnu.org/philosophy/why-free.html`, 2012.

[51] Marc Stiegler. An introduction to petname systems. `http://www.skyhunter.com/marcs/petnames/IntroPetNames.html`, February 2005.

[52] Daniel Stutzbach and Reza Rejaie. Improving lookup performance over a widely-deployed dht. In *INFOCOM*. IEEE, IEEE, 2006.

[53] Aaron Swartz. Squaring the triangle: Secure, decentralized, human-readable names. `http://www.aaronsw.com/weblog/squarezooko`, January 2011.

[54] S. Thomson, C. Huitema, V. Ksinant, and M. Souissi. Rfc 3596: Dns extensions to support ip version 6. Technical report, Network Working Group, October 2003.

[55] Alexander Ulrich, Ralph Holz, Peter Hauck, and Georg Carle. Investigating the OpenPGP Web of Trust. In *16th European Symposium on Research in Computer Security (ESORICS 2011)*, LNCS. Springer Verlag, September 2011.

[56] Anne van Kersteren. Cross-origin resource sharing. Technical report, W3c Working Draft 3, `http://www.w3.org/TR/cors/`, April 2012.

[57] T. van Leijenhorst, D. Lowe, and K-W Chin. On the viability and performance of dns tunneling. In *The 5th International Conference on Information Technology and Applications*, June 2008.

[58] Chih-Chiang Wang and Khaled Harfoush. Shortest-path routing in randomized dht-based peer-to-peer systems. *Comput. Netw.*, 52(18):3307–3317, 2008.

[59] Klaus Wehrle, Stefan Götz, and Simon Rieche. *Distributed Hash Tables*, volume 3485 of *Lecture Notes in Computer Science*, chapter 7. Springer, 2005.

[60] Zooko Wilcox-O'Hearn. Names: Decentralized, secure, human-meaningful: Choose two. `http://zooko.com/distnames.html`, Jan 2006.