Web-Based User Interface for a Simple Distributed Security Infrastructure (SDSI)

by

Gillian D. Elcock

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

Master of Engineering in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 1997

© Gillian D. Elcock, MCMXCVII. All rights reserved.

The author hereby grants to MIT permission to reproduce and distribute publicly paper and electronic copies of this thesis document in whole or in part, and to grant others the right to do so.

Author ....
Department of Electrical Engineering and Computer Science
May 23, 1997

Certified by ....
Ronald L. Rivest
Edwin Sibley Webster Professor of Computer Science and Engineering
Thesis Supervisor

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

Web-Based User Interface for a Simple Distributed Security Infrastructure (SDSI)

by

Gillian D. Elcock

Submitted to the Department of Electrical Engineering and Computer Science
on May 23, 1997, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Electrical Engineering and Computer Science

Abstract

Public-key certificates are becoming increasingly important in today's electronic age. In order for public-key certificates to be deployed in a large, distributed community such as the Internet, there needs to be a public-key infrastructure that allows the public-keys of others to be found securely. SDSI (Simple Distributed Security Infrastructure) is a recent proposal for a public-key infrastructure that attempts to address the problems of some of the existing schemes. This thesis is part of the work involved in an initial implementation of SDSI. It presents a World Wide Web-based user interface that allows users to interact with their SDSI servers and public-key certificates to carry out many of the functions described in the SDSI proposal.

Thesis Supervisor: Ronald L. Rivest
Title: Edwin Sibley Webster Professor of Computer Science and Engineering

Acknowledgments

I would like to thank the following people:

- Professor Ron Rivest, my thesis advisor, who provided guidance on all parts of this thesis, and gave me a great deal of his time, especially during that last fateful week before the thesis was due. Thank you for giving me the opportunity to work with you.
- Be Blackburn, for her sunny smile and constant supply of chocolates.
- Professor Jerome Saltzer and Mitchell Charity, who gave me my first UROP, where I learned as much as I did in any of my classes.
- Professor Peter Szolovits, who was a wonderful academic advisor during my time at MIT.
- Anne Hunter, for always being there to answer questions with endless patience.
- Shiva Sean Sandy, for reading a near-complete draft of the thesis, and providing helpful feedback.
- All my friends.
- And of course my parents, brothers and sisters, and all my family in Trinidad, the United States, and Canada, for their constant support throughout my five years at MIT.

This research was supported by DARPA grant DABT63-96-C-0018.

Contents

1 Introduction **11**

2 Background **13**

2.1 Public-Key Cryptography 13

2.2 Digital Signatures 14

2.3 Traditional Public-Key Certificates and Infrastructures 14

2.3.1 Existing Infrastructures 16

2.4 Newer Ideas about Public-Key Infrastructures 16

3 SDSI **19**

3.1 Introduction 19

3.2 Description of Key Features 19

4 Design of the SDSI User Interface **29**

4.1 Goals 29

4.2 Type of Interface 31

4.3 Functionality 32

4.4 Design Choices 45

4.4.1 Structure 45

4.4.2 Limiting the UI 47

4.5 The Underlying Architecture 49

5 Implementation of the SDSI User Interface **51**

5.1 The Web Server 51

5.2 Interacting with the SDSI Shell (sdsish) 52
5.3 The Security Loop 54

6 Discussion and Future Work 59

6.1 Challenges 59
6.2 Future Improvements 60
6.3 The Evolution of SDSI 63

7 Conclusion 67

List of Figures

4-1 The SDSI Homepage 33

4-2 The Autocert 34

4-3 Search for Principals and Issue Certificates - Principal Found 36

4-4 Issue a Name/Value Certificate 37

4-5 Search for Principals and Issue Certificates - Certificate Found 38

4-6 List Certificates in Your Cache 40

4-7 Result of List Certificates in Your Cache 41

4-8 Defining a Group 43

4-9 Result of Defining a Group 44

4-10 The Underlying Architecture 49

5-1 The Underlying Architecture with Security Loop 56

Chapter 1

Introduction

Public-key certificates are becoming increasingly important in today's electronically linked world. They are essential for data encryption, digital signatures, secure electronic commerce, and a host of other applications. In order for public-key certificates to be deployed in a large, distributed community such as the Internet, there needs to be a public-key infrastructure that allows the public-keys of others to be found securely. While there are existing proposals for public key infrastructures, some of which are already used in practice, many of them are complex and inadequate, and to date, the development of a widely used public-key infrastructure has been slow.

SDSI (Simple Distributed Security Infrastructure, pronounced "sudsy") [8] is a proposal by Ronald Rivest and Butler Lampson that has been recently put forward in an attempt to address some of the problems of existing public-key schemes, as well as present some new ideas for implementing a large-scale public key infrastructure. It provides a simple design which uses linked local name spaces rather than a global, hierarchical one. It also specifies ways of defining groups and writing access control lists. This thesis is part of the work involved in an initial implementation of SDSI. It presents a World Wide Web-based user interface that allows users to interact with their SDSI servers and public-key certificates to carry out many of the functions described in the SDSI proposal. The SDSI user interface (UI) is built upon the library written by Matt Fredette as part of "An Implementation of SDSI" [5].

Chapter 2 of this thesis provides some background information about public-key

cryptography and certificates, which is needed to understand the theoretical basis for SDSI, and this thesis work. It also gives brief descriptions of some of the existing public-key infrastructure schemes, as well as some of the more recent proposals.

Chapter 3 presents an overview of SDSI, which is needed to provide a reference for the subsequent chapters.

Chapter 4 describes the design of the SDSI UI, and the reasons behind some of the design choices. It also contains diagrams of some of the screens that a SDSI user might encounter when using the UI.

Chapter 5 describes some of the key steps involved in implementing the user interface.

Chapter 6 gives a discussion of possible future expansions to this work, and also outlines some of the developments of the next version of SDSI. Chapter 7 ends with a conclusion.

Chapter 2

Background

This chapter gives a brief introduction to public key cryptography, public-key certificates, and some of the previous work that has been done in this area. A more detailed presentation of this material can be found in the book "Secure Electronic Commerce" [4] or in a recent article in "Byte" [9].

2.1 Public-Key Cryptography

If two parties want to exchange information secretly, the sender can transform the data using a cryptographic algorithm through a process called *encryption*. The data that is input to the process is called *plaintext*, and the output is called *ciphertext*. The receiver can invert the encryption by a decryption process, to retrieve the original plaintext. Both encryption and decryption require a key as a parameter. In a symmetric encryption scheme, the same key is used for both encryption and decryption. This key is kept secret from everyone except the sender and receiver. In an asymmetric or public-key encryption scheme, two different but mathematically related keys are used for encryption and decryption. One, called the private key, is kept secret, known only to its owner. The other key is made known publicly, hence the term *public-key cryptography*. Data can be encrypted with the private key and decrypted with the corresponding public key, and vice versa.

2.2 Digital Signatures

Public-key cryptography can be used to create digital signatures of messages. A user can digitally sign data by encrypting it with the private key of his public-private key pair. This signature is sent along with the message. Anyone else can then use the public-key of that user to decrypt the signed data, and check that the result of the decryption and the message that was sent, are equal. Usually, for efficiency, instead of encrypting the actual message, a hash of the message is computed, and the resulting hash value is signed. The verifier can then decrypt the hashed message with the signer's public key, independently compute a hash of the message (using the same hash function as the signer), and then check whether these two results are equal. If they are, he knows that the owner of the public-key signed the message, and that the message was not changed in any way during transmission.

2.3 Traditional Public-Key Certificates and Infrastructures

This section describes the traditional model of public-key infrastructures. It also gives a brief description of two schemes that essentially use this model, X.509 and PGP.

A public-key certificate is a data structure that has been traditionally used to bind a public-key to a particular individual, organization, or process. In this model, a public-key certificate is digitally signed by the issuer of the certificate, i.e. the person or entity that has confirmed the binding between the public-key and the holder of the certificate.

When any entity wishes to verify the digital signature of another, it first needs to obtain a copy of the public-key of the entity that made the signature. When this occurs, it is important that the person obtaining the public-key is certain that this public-key is actually the correct one for the party whose signature he is trying to verify. If, for example, an adversary could substitute his own public-key for another

one, then that adversary could sign a message with his private key, claim the message to be from someone else, and spoof the verifier into using his public-key for signature verification. Then the verifier would accept a message from the adversary as coming from someone else. Therefore, the whole security of digital signatures depends on the binding of public-keys to the correct entities.

This problem can be easily solved among a small number of users who know each other. Two people who want to exchange public-keys securely can do so, for example, by swapping diskettes which hold their public-key values. As the set of users becomes large and dispersed, however, this model becomes impractical. Public-key certificates provide a solution to public-key distribution that is scalable and secure.

In order to use public-key certificates to obtain a user's public-key, a verifier must obtain and validate that user's public-key certificate. One validates a certificate by verifying the signature of the issuer of that certificate. This transforms the problem of acquiring the public-key of a user into one of acquiring the public-key of the issuer of that user's certificate. The issuer will also have a public-key certificate, and so the process of certificate validation can be used recursively to develop a chain of signatures. This process continues until the verifier finds a certificate signed with a public-key that he knows and trusts. Then he can unwind the chain of certificates by verifying the signatures on the certificates in the reverse order in which they were obtained, until he can verify the certificate of the original user.

We have seen that public-key cryptography and digital signatures are essential for the development of secure electronic communication. Public-key certificates are the tools necessary to apply public-key technology. Public-key infrastructures (PKI)s are the supporting structures that are needed to enable public-key technologies to be used on a wide scale. When one tries to use and manage public-key certificates in the real world, especially in very complex and diverse environments, many technological, legal and other problems arise. These issues need to be addressed in order to see public-key technologies used to their fullest potential. To date, the development of a wide-spread public-key infrastructure has been very slow.

2.3.1 Existing Infrastructures

X.509. X.509 certificates [7] are the most widely used today. Their use includes the MasterCard/Visa Secure Electronic Transaction (SET) infrastructure for Internet-based bank card payments, the United States Federal Government, and the ICE-TEL initiative in Europe. X.509 is based on the use of designated certification authorities (CA)s that have the authority to sign public-key certificates for other entities, verifying that this entity is indeed the holder of a certain public-key. X.509 also attempts to assign a global "distinguished name" (DN) for individuals, organizations, and applications. As an example, a DN might consist of the following: {`C` = US, `S` = Massachusetts, `L` = Cambridge, `O` = ABC Inc., `CN` = Bob Jones}, where `C` represents the country, `S` the state name, `L` the locality name, `O` the organization name, and `CN` the common name of the individual. However, the recent X.509 version 3 standard allows the use of other local names in certificates, mainly out of a realization that the DN naming scheme was too constraining.

Pretty Good Privacy (PGP). Philip Zimmermann's PGP [10] is popular among Internet users for encrypting and signing email. Each user chooses a name for his public-key, which is usually his full name followed by an email address, such as `Bob M. Jones <bob@abc.com>`. Any user can sign the public-key of another, vouching that this is indeed that person's correct public-key. Users can also make decisions about whether they trust other people to act as "introducers" for other PGP public-keys, and how much they trust them. If the level of trust of the introducers of a certain key reaches a certain threshold, that key will be accepted as valid. This scheme allows users to employ a grass roots, "web of trust" method for obtaining and issuing public-key certificates.

2.4 Newer Ideas about Public-Key Infrastructures

This section briefly presents some of the newer public-key certificate schemes that have been proposed. Most of them move away from the traditional idea of binding

a public-key to a particular individual or entity by using that individual's "unique" name.

Simple Public Key Infrastructure (SPKI). Proposed by the IETF working group, and in particular, Carl Ellison, SPKI [3] is a recent (June, 1996) proposal for a public-key infrastructure. SPKI certificates grant specific authorizations to the subject of a certificate, which is a public-key, rather than an individual or other entity. An example of such an authorization would be to allow a particular public-key to *telnet* to a certain host using a particular username.

Simple Distributed Security Infrastructure (SDSI). SDSI is the joint work of Ronald Rivest and Butler Lampson. The SDSI proposal [8] (Sept., 1996) outlines a public-key certification scheme that is simple and flexible, employing completely local names, and using group definitions for access control. It is described in more detail in Chapter 3.

PolicyMaker. Proposed by Blaze, Feigenbaum and Lacy (May, 1996), PolicyMaker [2] is not strictly a public-key certificate scheme, but rather provides a generic language for expressing trust relationships, policies and credentials, so that decisions can be made about requests based on digitally signed messages.

As it can be seen from the descriptions above, there is still a lot of ground-breaking research being done in this area, and there are many unanswered questions about what the "right" approach is to solving the problem of deploying a secure, wide-spread, easy to use public-key infrastructure.

Chapter 3

SDSI

3.1 Introduction

This chapter gives an overview of the first version of SDSI, SDSI 1.0, which is the basis for the implementation work of this thesis. Much of it is derived from the paper "SDSI - A Simple Distributed Security Infrastructure"[8].

SDSI's design provides a simple public-key infrastructure which uses linked local name spaces rather than a global, hierarchical one. It also specifies ways of defining groups and writing access control lists. Part of the motivation behind SDSI's creation was a realization that while the traditional focus of public-key certificates has been on name-key bindings, the real need is to have a way to build secure distributed systems. Within this framework, *access control* is often the main problem i.e. deciding whether a certain digitally signed request for a certain resource (e.g. an HTTP request for a web page) should be granted. Given these facts, one of the aims of SDSI is to provide access control lists (ACL)s that are easy to create and maintain. The following section expands on the key points of SDSI.

3.2 Description of Key Features

Principals are Public Keys. Public-keys are central in SDSI. The idea of an individual (person, process, or machine, etc) is not required, though of course such

individuals will control the private key associated with the public-key. However, the most important thing about a principal is its ability to verify signed statements, and this is why each SDSI principal is defined by its public key.

SDSI objects are represented by *S-expressions*, and so a public-key might be represented by the following data structure:

```
(Public-Key:
  ( Algorithm: RSA-with-SHA1 )
  ( N: =Gt802Tbz9HKm067= )
  ( E: #11 ) )
```

where the `Algorithm:` field specifies the algorithm that should be used to verify the public-key (in this case RSA with the SHA1 hash function), `N:` gives a base-64 encoding of the RSA public modulus, and `E:` gives the hexadecimal value of the RSA exponent.

A SDSI principal is made up of such a public-key along with one or more optional Internet addresses. A principal might be represented by the following:

```
(Principal:
  ( Public-Key:
     ( Algorithm: RSA-with-SHA1 )
     ( N: =Gt802Tbz9HKm067= )
     ( E: #11 ) )
  ( Principal-At: "http://www.abc.com/cgi-bin/sdsi-server")
  ( Server-At: "http://www.xyz.com/cgi-bin/sdsi-server" ) )
```

The `Principal-At:` and `Server-At:` fields specify Internet addresses where queries to the principal can be addressed, or certificates can be distributed on behalf of the principal.

Every principal is a CA. One of the principal features of SDSI is that all public-keys are equal. There is no hierarchical global infrastructure (though in practice some principals might become more important than others). In particular, SDSI certificates

can be created and signed by any principal, not just certain designated certification authorities, or "CA"s.

Local Name Spaces. Each principal in SDSI can create a local name space in which it can bind other principals to any names it chooses. These names can be arbitrarily chosen, perhaps derived from nicknames, email addresses, account numbers, etc. Some example local names might be:

```
alice
bob-jones
account-12345
```

There is no fixed global name space in which each principal must be linked to a unique name. The principal one user refers to as `bob-jones` may be different from the principal another user refers to as `bob-jones`.

Principals assign values to the local names in their name spaces by issuing certificates, which are described in more detail later. The policies that a principal follows when issuing certificates are totally up to that principal.

Linked Local Name Spaces. One of the most important contributions of SDSI is the idea of linked name spaces. Though there is no global name space in SDSI, principals can conveniently link their local name spaces in order to reach principals in other name spaces. Each principal can export to others its binding of other principals to names by issuing name/value certificates. Thus, if a user's local name `bob` refers to some principal, then that user can refer to the principal that `bob` calls `alice` as

```
(ref: bob alice )
```

or, using syntactic sugar,

```
bob's alice
```

Bob exports his binding by issuing a name/value certificate which binds his local name `alice` to alice's principal. This concept can be extended to longer references such as

`bob's alice's eve`, which is the syntactically sugared version of `(ref: bob alice eve)`. This reference is well-defined for a user if he has bound `bob` to some principal, and that principal has in turn bound `alice` to another principal, which has in turn bound `eve` to a third principal. Other examples of such *extended references* are:

```
Visa's account-123456789
mit's lcs's rivest
Harvard's faculty's joan
```

There can also be symbolic definitions in SDSI. For example, a user's local name `bob-jones` can be defined to be `mit's bob-jones`, and so if `mit` changes the principal it calls `bob-jones`, then the principal that user calls `bob-jones` will change as well.

Given these capabilities, a typical SDSI user will only need to obtain maybe 5-20 public-keys and then obtain other principals by linking off these. However, the process of accepting these initial 5-20 public-keys and binding them to local names is very important, and should be done carefully and manually, making sure of the correct public-key owner, and choosing a meaningful local name.

Certificates. There are four main types of certificates: *Name/Value, Membership, Delegation,* and *Autocert.*

Name/Value Certificates. These certificates are the ones used to bind principals to local names, thereby exporting those bindings to the world. These certificates must be signed by the issuer, using its public-key principal. An example certificate, which binds a principal to the local name `alice-brown` is shown below:

```
(Cert:
  ( Local-Name: alice-brown )
  ( Value:
    ( Principal:
       ( Public-Key:
         ( Algorithm: RSA-with-SHA1 )
```

```
        ( N: =Gt802Tbz9HKm067= )
        ( E: #11 ) )
    ( Principal-At: "http://www.abc.com/cgi-bin/sdsi-server")
    ( Server-At: "http://www.xyz.com/cgi-bin/sdsi-server" ) )
  )
  ( Description: "Alice-Brown is a friend of 5 years")
  ( Signed: ...) )
```

The `Signed:` field will contain the signature on the certificate, with the date of signing, and an optional expiration date for the certificate. In addition, the actual signing principal may be appended to the signature.

As described before, certificates may symbolically bind a local name to another name. So, to define `bob-jones` to be `mit's bob-jones` the following certificate would be issued:

```
(Cert:
  ( Local-Name: bob-jones )
  ( Value: mit's bob-jones )
  ( Signed: ... ) )
```

Delegation Certificates. These certificates are used to authorize a group (of servers) to be able to sign certificates on behalf of the signing principal. Delegation certificates are meant to be used by SDSI servers to show that their signatures are authorized by other principals, but there are also many other possible uses for them. Delegation certificates have the following form:

```
(Delegation-Cert:
  ( Template: form )
  ( Group: group )
  ( Signed: ... ) )
```

This certificate authorizes every member of `group` to be able to sign objects matching `form`. To give a more concrete example, the following certificate:

```
(Delegation-Cert:
  ( Template: (Membership.Cert: (Group: math_lovers ) ) )
  ( Group: server-group )
  ( Signed: ... ) )
```

gives the group `server-group` the authority to sign membership certificates for the group `math_lovers`.

Membership Certificates. These are certificates that give principals membership in a particular SDSI group. They will be described along with groups in a later section.

The Autocert. An auto-certificate or Autocert is a special kind of certificate. It is signed by the principal that the certificate is about, and has the type `Auto-Cert:`. Every SDSI principal is required to have an Autocert. The only mandatory field is the `Public-Key` field, but it may contain other information as well. Most of these fields are there primarily for humans to read, and may be added at the user's discretion. Since all of this information is signed by the principal itself, it should not be trusted without suitable corroboration. An example of some of the fields an Autocert might contain is given below:

```
(Auto-Cert:
   ( Public-Key: ... )
   ( Principal-At: "http://www.abc.com/cgi-bin/sdsi-server")
   ( Server-At: "http://www.xyz.com/cgi-bin/sdsi-server" )
   ( Local-Name: alice-brown )
   ( Description: "I am an employee of ABC, Inc." )
   ( Phone: 617-246-3579 )
   ( Email-address: alice@abc.com )
   ( Signed: ... ) )
```

In this case the `Local-Name:` field gives the principal's favorite nickname for herself, which others may or may not use for this principal in their own name spaces. The `Description:` field gives arbitrary text by the entity that controls the public-key about herself.

Groups and ACLs. Another important feature of SDSI is its group definition capability. A SDSI group is a set of principals, and each group has a name which is local to some principal that is the "owner" of the group. Each SDSI principal can define groups, and only the owner of the group can change the group definition. The definition can list the group's members as explicit principals, or by name, or it can define the group in terms of other groups. Some example group definitions are given below:

```
( Group:  bob alice fred )
( Group: ( AND: friends over-18 ) )
( Group: mit's faculty staff ( Principal: ... ) )
( Group: ( NOT: enemies ) )
```

Another simple group is the group containing everyone, which is denoted by `ALL!`.

To actually bind a group to a local name and export the definition, a name/value certificate must be issued for the group. As an example:

```
(Cert:
   ( Local-Name: math_lovers )
   ( Value: ( Group: bob chris joan ) )
   ( Signed: ... ) )
```

Membership in a group can also be given to a principal without making an explicit group definition by using membership certificates. The certificate below gives the indicated `Principal` membership in the group `friends`:

```
( Membership.Cert:
  ( Member: ( Principal: ... ) )
```

```
  ( Group: friends )
  ( Signed: ... ) )
```

Groups are useful for creating access control lists, or "ACL"s. ACLs are used to describe who is authorized to access certain data (e.g. a file) or perform certain operations (e.g connect to a web server). Usually this authorization is done by defining a group of authorized principals, and then placing the group name on an appropriate ACL. This method is especially efficient when the same group of principals is authorized on many different ACLs. Then to change authorizations, the group definition can be updated with a single modification, without having to update the ACL. Also, groups can be given meaningful names so that writing ACLs becomes simpler. Each principal can define its own groups and export its definitions to others. Some groups that might be useful in ACLs are given below:

```
friends
mit's chemistry-department's faculty
USA's over-18
```

ACLs are made up of a `type` and `definition`, where `type` determines the operation that is being controlled (e.g. `read`) and `definiton` is the group that is being given the authority. As an example, the following certificate can only be read by the group `math_lovers`:

```
( Cert:
   ( Local-Name: alice-brown )
   ( Value: ( Principal: ... ) )
   ( ACL: ( read: math_lovers ) )
   ( Signed: ... ) )
```

One ACL of special interest is `(ACL: (read:  ALL!) )` which allows anyone to read the associated object.

On-line Internet orientation. SDSI assumes that principals that issue certificates can provide an on-line Internet service, or arrange to have one provided via a

designated server or servers. Each SDSI user will have a *local cache* which contains all of his certificates, and his SDSI servers will have a mirror image of this collection. A SDSI principal may sign certificates off-line and then have his servers distribute them upon request. These servers would respond to and make queries, as described in the next section. SDSI servers should have high reliability and on-line availability.

Queries. Queries are carried out and responded to by SDSI servers. The three types of queries are *Get*, *Membership*, and *Reconfirmation*.

Get Queries. `Get` queries are the primary way by which the certificates of other principals can be obtained. These certificates export the name/value bindings of that principal and allow the linking of local name spaces to occur as described previously. The `Get` query function is therefore a central one in SDSI, and it is carried out by SDSI servers. Servers hold a database of certificates, and they can be queried to return certificates that satisfy some criteria. The `Get` query always contains a `To` field which specifies a principal, and the certificates that are returned must have that principal as the primary signer. A `Get` query also specifies a `Template` which gives the form of the desired certificates. Therefore, to get a certificate with the local name `jim` issued by the indicated `Principal`, the following query would be composed:

```
(Get.Query:
  ( To: ( Principal: ... ) )
  ( Template: ( Cert: ( Local-Name: jim ) ) )
  ( Signed: ... ) )
```

Membership Queries. Membership queries are used to obtain membership certificates. A principal can query a server to find out whether or not it is a member of a particular group, and the server can respond with a membership certificate. The reply to a membership query is either `true`, `false`, or `fail`. An optional `Hint:` may be given in the `fail` case, explaining to the client how it can supply credentials (for example membership certificates for other groups) that would eliminate the failure

the next time.

Reconfirmation Queries. SDSI does not have "certificate revocation lists" as a means of revoking the signature on a previously signed object. Instead, signatures may be designated as needing periodic reconfirmation. The signer can specify the reconfirmation period that is appropriate for that signature: some signatures might need to be re-confirmed yearly, while others might need reconfirmation hourly. SDSI servers can make reconfirmation queries to find out whether the signature on a particular object has been reconfirmed.

Chapter 4

Design of the SDSI User Interface

The design and implementation of the SDSI UI was an iterative process, and in the end, the design was re-worked several times. Initial implementations pointed out certain flaws in the original design, so the design would be changed, which would require a new implementation, and so on. This was somewhat to be expected given that this user interface was the first of its kind for SDSI, which was a totally new proposal, and the process proved to be a valuable learning experience. This chapter describes the final design that was implemented in this thesis. Further re-working is expected in order to produce a completely finished user interface.

4.1 Goals

The SDSI UI is meant provide each user with a way to manage his SDSI servers and certificates. This will include functions such as creating certificates, querying other servers for certificates, viewing certificates in his local cache, and editing his Autocert.

There are many different people who may want to use public-key certificates. The range of sophistication of the user may go from a system administrator in charge of the security of a large corporation, all the way to a person who casually uses public-key certificates to authenticate email from friends.

The aim of this user interface is to make SDSI accessible to an average person who has a basic knowledge of public-key cryptography, not just cryptographers and others

with advanced knowledge. It is envisioned that as time passes, a typical SDSI user will accumulate many name/value certificates. After manually accepting the public keys of a few individuals or entities (maybe 5-20 of them) and associating them with local names, the user will be able to use SDSI's capability of linking name spaces to obtain other certificates.

As an example, we consider the user Alice who has just enrolled as a student at MIT. When Alice logs into her student account she generates a SDSI public-key, and can start up the SDSI UI, using the password which protects the private part of her key.[1]

One of the first things Alice will want to use the UI for is to create her Autocert, which can contain information about herself in addition to her public-key.

Alice soon decides that she wants to be able to receive authenticated messages from her high-school friend Bob, who now goes to Harvard. In order to do this, she needs Bob's public-key, and so Alice and Bob decide to exchange public-keys by placing their SDSI principals on diskettes and swapping them. Once she has the diskette with Bob's key, and places the key in a file, Alice can load it in through the UI and issue a certificate for Bob's key, perhaps giving it the local name *MyBob*. Alice can also issue membership certificates for Bob's principal, for any group she decides to create.

Next, Alice meets Chris through Bob, and wants to get his public-key. Bob emails Alice, telling her that he has a certificate for Chris in his cache, under the local name `Chris-Smith`. Alice has trust in Bob's judgement with respect to binding public-keys to the correct entity. Therefore she decides to query Bob's SDSI server for Chris's certificate. Her query will be input through the SDSI UI as `MyBob's Chris-Smith`. Alice can then issue a certificate for the public-key principal that is returned by the query, again using the SDSI UI. She can chose any local name she wants for this principal, perhaps `Chris-Smith` or `MyChris`.

[1]MIT currently uses X.509 public-key certificates to allow students to view their financial and academic information via the World Wide Web.

Alice soon makes friends at MIT, and in particular she wants to obtain the public-key of Eve, another student. Alice already has MIT's public-key principal and has given it the local name `MIT`, and she knows that MIT has a certificate for Eve with the local name `Eve25`. So, because she has faith in the reliability of MIT's SDSI servers, she decides to create a certificate for Eve with the local name `MyEve`, binding it symbolically to the value `MIT's Eve25`.

As a final example, Alice wants to create a group called `math_lovers`, and she plans to make certain files available for reading to the principals that are members of this group. She can define the group using the SDSI UI.

The SDSI UI should enable Alice (or any other user) to perform all of the functions listed above, as well as the other basic SDSI functions described in the SDSI proposal [8]. In addition, the UI should be simple and easy to use, and should provide the user with ample on-line support.

It should be noted that the SDSI UI is built upon the SDSI library written by Matt Fredette as part of "An Implementation of SDSI"[5].

4.2 Type of Interface

An early design choice was to make the SDSI UI web-based. This requires each user to have a World Wide Web browser that supports forms, such as Netscape. The reasons for this choice are:

Familiarity to users. The World Wide Web is fast becoming one of the most widely used services of the Internet. Web browsers such as Netscape provide a very familiar interface to users, and will help contribute towards fulfilling the goals of ease of use and accessibility of the SDSI UI.

Functionality. Most web browsers provide functionality such as image and postscript viewing. This capability is useful because some SDSI certificates may contain image files.

Availability. Web browsers are widely available and easily obtained, often for free. This availability will make a web-based user interface accessible to a large number of users on many different machine platforms.

Fits well with SDSI. As described previously, SDSI is designed with the intent that principals will have highly available on-line servers that can make and respond to queries. Having a web-based user interface seems to fit well with this Internet orientation.

4.3 Functionality

This section describes how some of the pages of the SDSI UI work, and also gives diagrams of these pages. It continues with the example of Alice and her friends, showing how Alice would use the UI to accomplish her tasks.

The Autocert. When Alice first starts up the SDSI UI she encounters her `SDSI Homepage`. Figure 4-1 contains a diagram of this first screen. Alice knows that she needs to create an Autocert, so she goes first to the `Manage Your Autocert` page. When this page appears, it displays Alice's public-key and gives her space to enter other name/value fields that she would like to include. Figure 4-2 shows this screen, along with some example information that Alice might enter about herself:

```
Email-address: alice@mit.edu
Description: "I am a first year student at MIT. I plan to major
in Math."
```

When Alice has completed all the fields she wants, she can click on the `Sign` button to sign the Autocert and add it to her cache of certificates. The next time that Alice enters the `Manage Your Autocert` page, it will display the information that she entered.

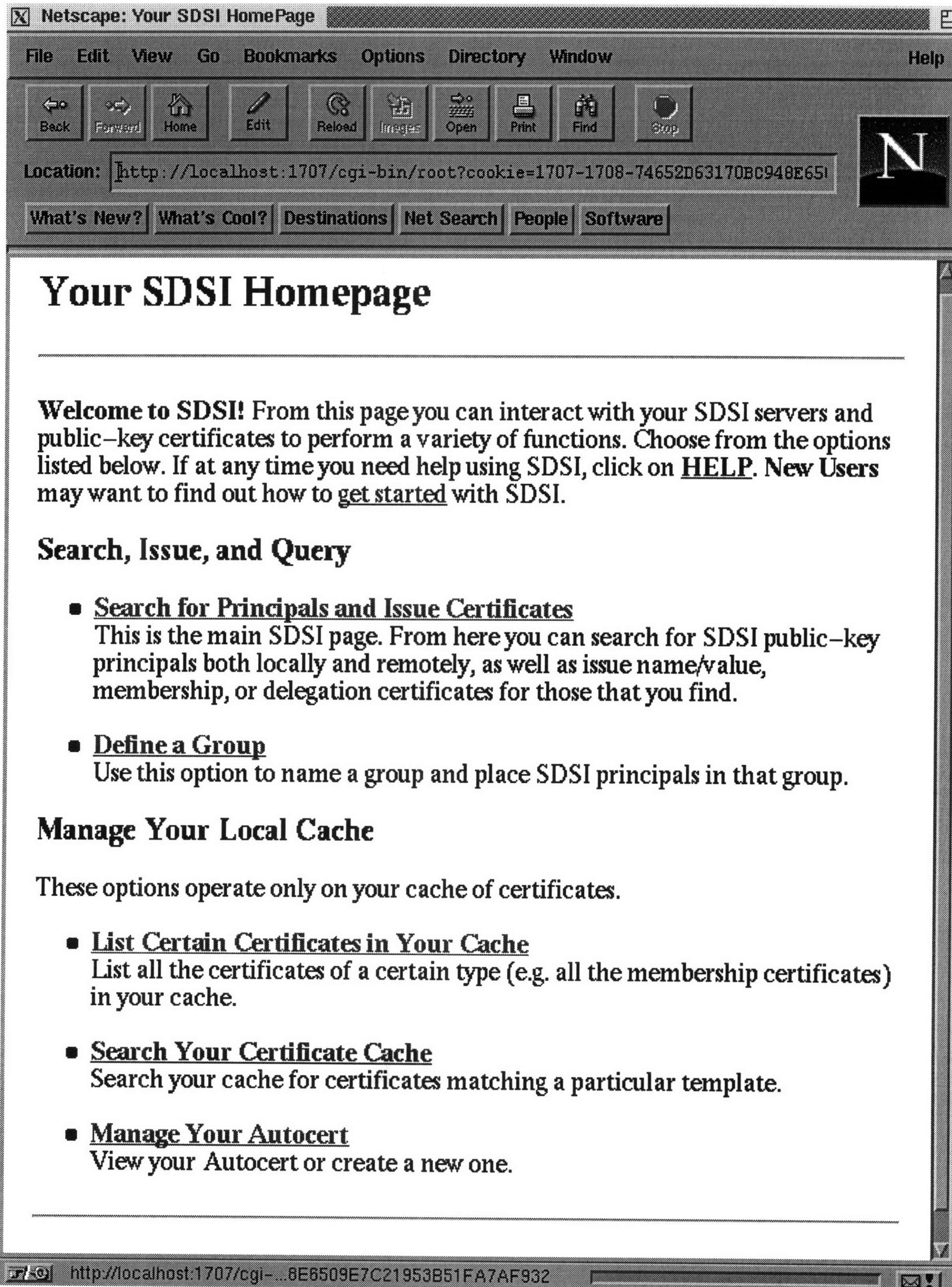

Figure 4-1: The SDSI Homepage

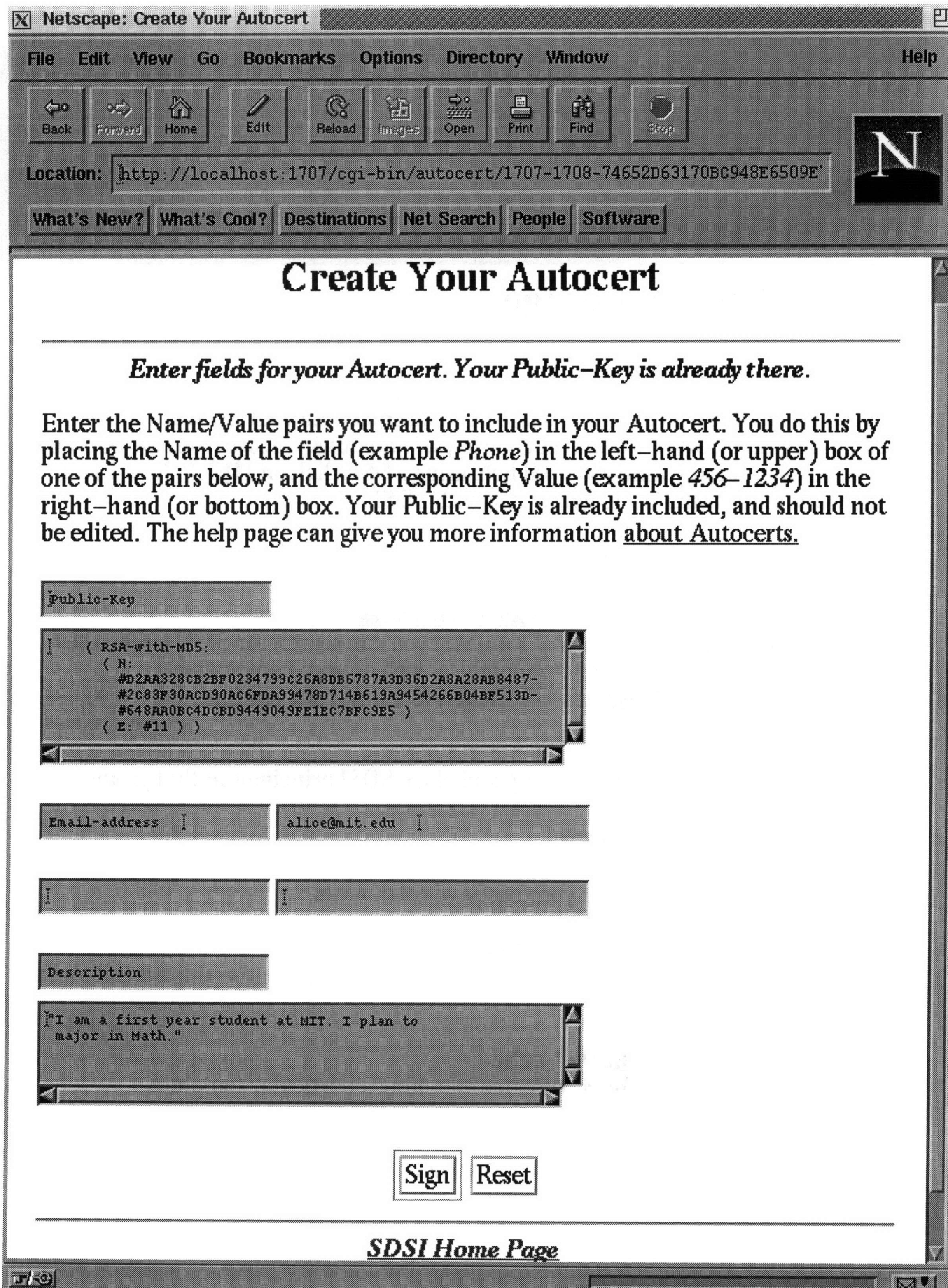

Figure 4-2: The Autocert

Loading Principal from File. Alice then obtains a public-key principal from her friend Bob, and places it in a file called `bob_key`. She wants to use this principal to issue a certificate for Bob, so she goes into the `Search for Principals and Issue Certificates` page of the SDSI UI. There she goes to the `Search for Principals` section, selects the `Filename` option and types in the filename `bob_key`. Clicking on `Submit` loads in the principal from the file and displays the results to Alice. Figure 4-3 shows the appearance of this screen.

Issue Name/Value Cert. Alice wants to continue by issuing a name/value certificate for Bob's public-key principal, so she goes to the `Issue Certificates` section of the `Search for Principals and Issue Certificates` page and clicks on the `Issue Name/Value Cert` link. This takes Alice to a page where she can enter the local name that she wants to bind Bob's principal to, as well as an expiration date for the certificate. Figure 4-4 shows this page while she is in the process of issuing a certificate with the local name `MyBob` and expiration date `December 31, 1998`. If Alice clicks on the `Sign` button, a certificate with these attributes will be signed by her principal, and added to her cache.

Issue Membership Cert. After a while, Alice learns about SDSI groups and membership certificates, and she decides she wants to create a group called `friends`. However, at this point she still considers Bob to be her only friend, so she decides to issue a membership certificate for Bob. To do this she once again goes to the `Search for Principals and Issue Certificates` page. There she selects the `SDSI Name` option, and enters her local name for Bob's principal, `MyBob`. This time a certificate is found, as shown in Figure 4-5. Alice will then choose the `Issue Membership Cert` option, and enter information on the resulting page about the name of the group, and also an expiration date for the certificate, similar to the name/value case in Figure 4-4.

Get Query. When Alice meets Chris later on, and wants to query Bob's server for his principal, she can once again use the `Search for Principals and Issue Certificates` page. Again, her query will also use a `SDSI Name`, but this time it will

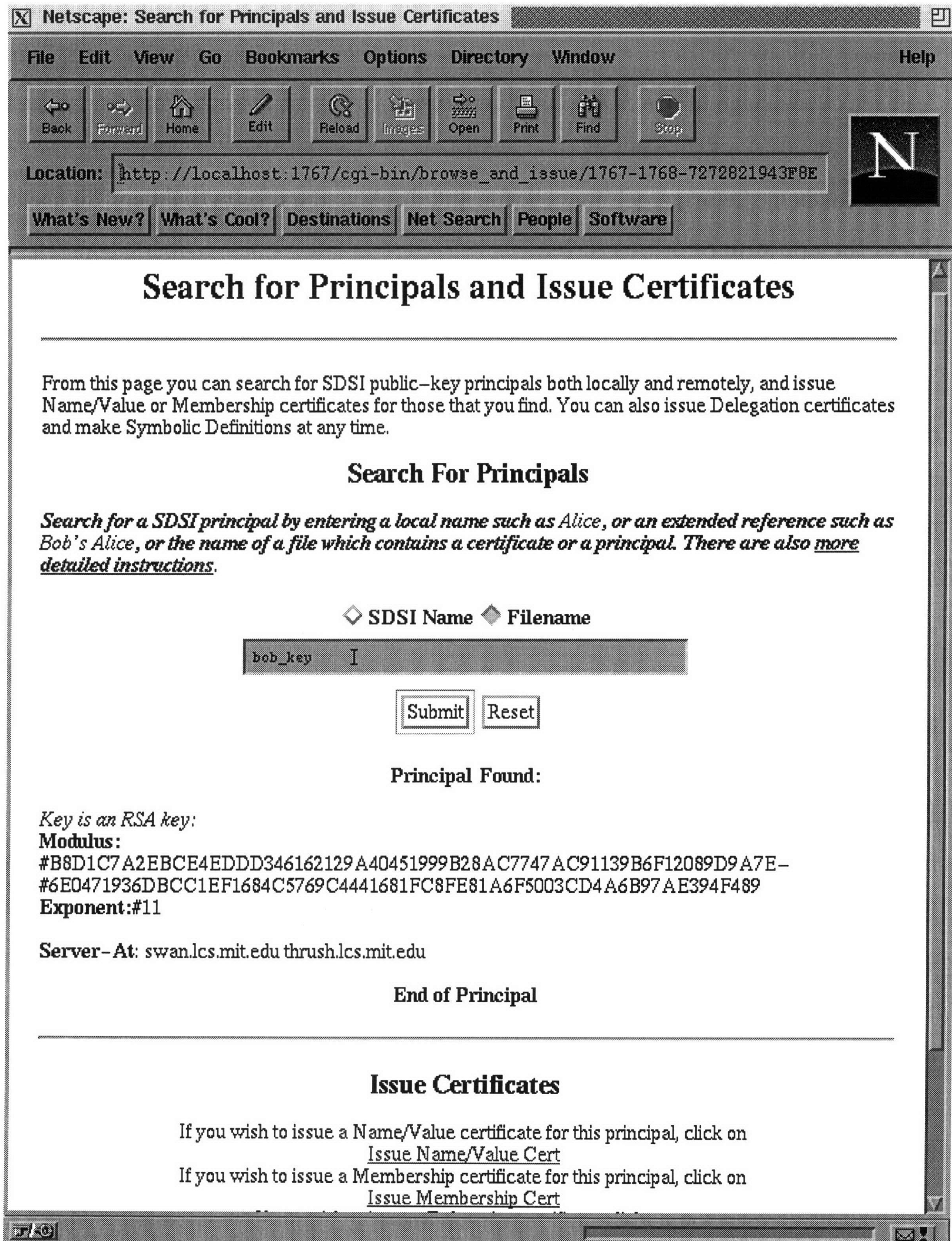

Figure 4-3: Search for Principals and Issue Certificates - Principal Found

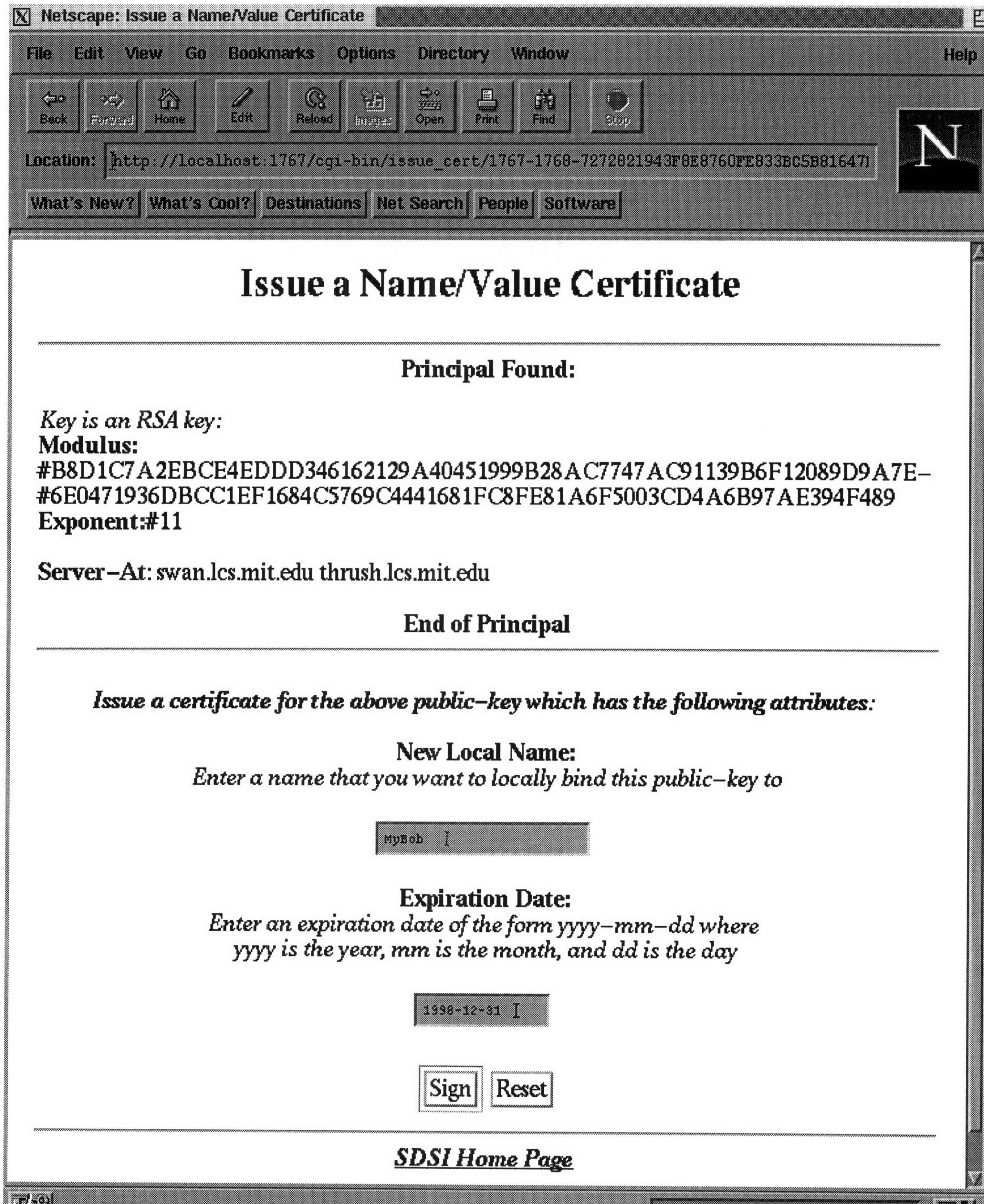

Figure 4-4: Issue a Name/Value Certificate

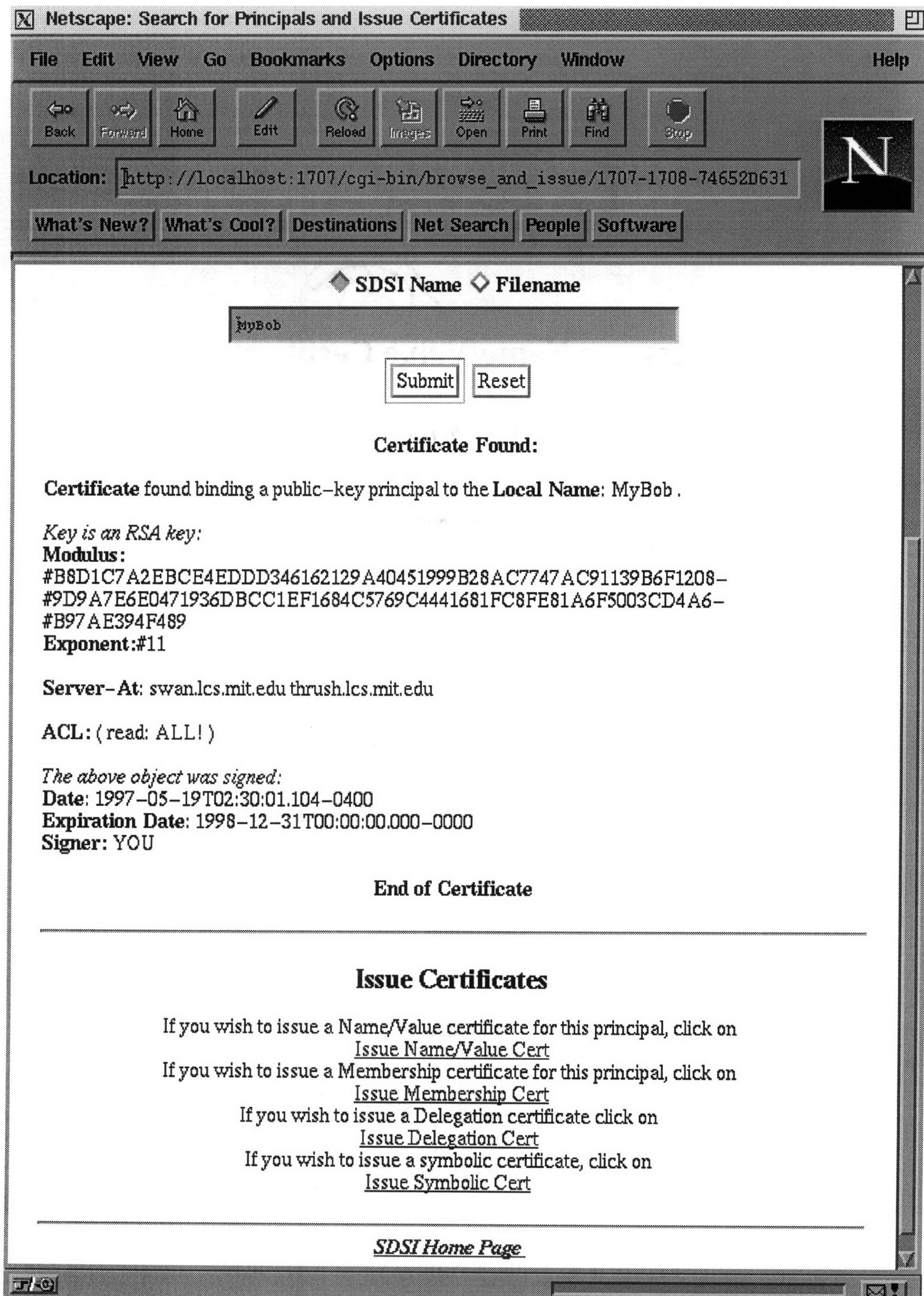

Figure 4-5: Search for Principals and Issue Certificates - Certificate Found

be the extended reference MyBob's Chris-Smith. If Bob has a certificate binding Chris' principal to the local name Chris-Smith, then the query will be successful, and a certificate will be displayed, in a similar fashion to the one in Figure 4-5. Alice can then issue name/value or membership certificates for Chris' public-key principal as she chooses. For the rest of this example, we assume that she has issued a name/value certificate for his principal, with the local name MyChris.

Symbolic Definition. Recall that Eve is another MIT student that Alice makes friends with. Alice wants to create a certificate for Eve also, but instead of explicitly searching for Eve's principal, she decides to symbolically define Eve to be MIT's Eve25. This principal will be well defined if Alice has a certificate which she signed which binds MIT's public-key principal to the local name MIT, and if this MIT principal has a certificate binding Eve's principal to the name Eve25. To make the symbolic definition, Alice will once again go to the Search for Principals and Issue Certificates page, but she will be able to go straight to the Issue Certificates section and click on the Issue Symbolic Cert link (without doing any searching). This page will give her the option to enter a local name, (such as MyEve), the value MIT's Eve25, and an expiration date.

List Certificates in Cache. As the semester progresses, Alice issues more and more certificates for the people she meets. She cannot always keep track of all of them, and one day in particular, she wants to see how many membership certificates she has issued. To do this she can go to the List Certain Certificates in Your Cache page from her SDSI Homepage. On this page, she would select Membership, and then click on the Submit Query button. Figure 4-6 shows this page, and Figure 4-7 shows the result of this search. Clicking on one of the links shown in Figure 4-7 will display the fields of the corresponding certificate which, in this case, will include the group name and the expiration date of the membership certificate.

Group Definition. As a final example, Alice decides to define a SDSI group. She has discovered that Bob, Chris and Eve all share her love for math, and so she decides

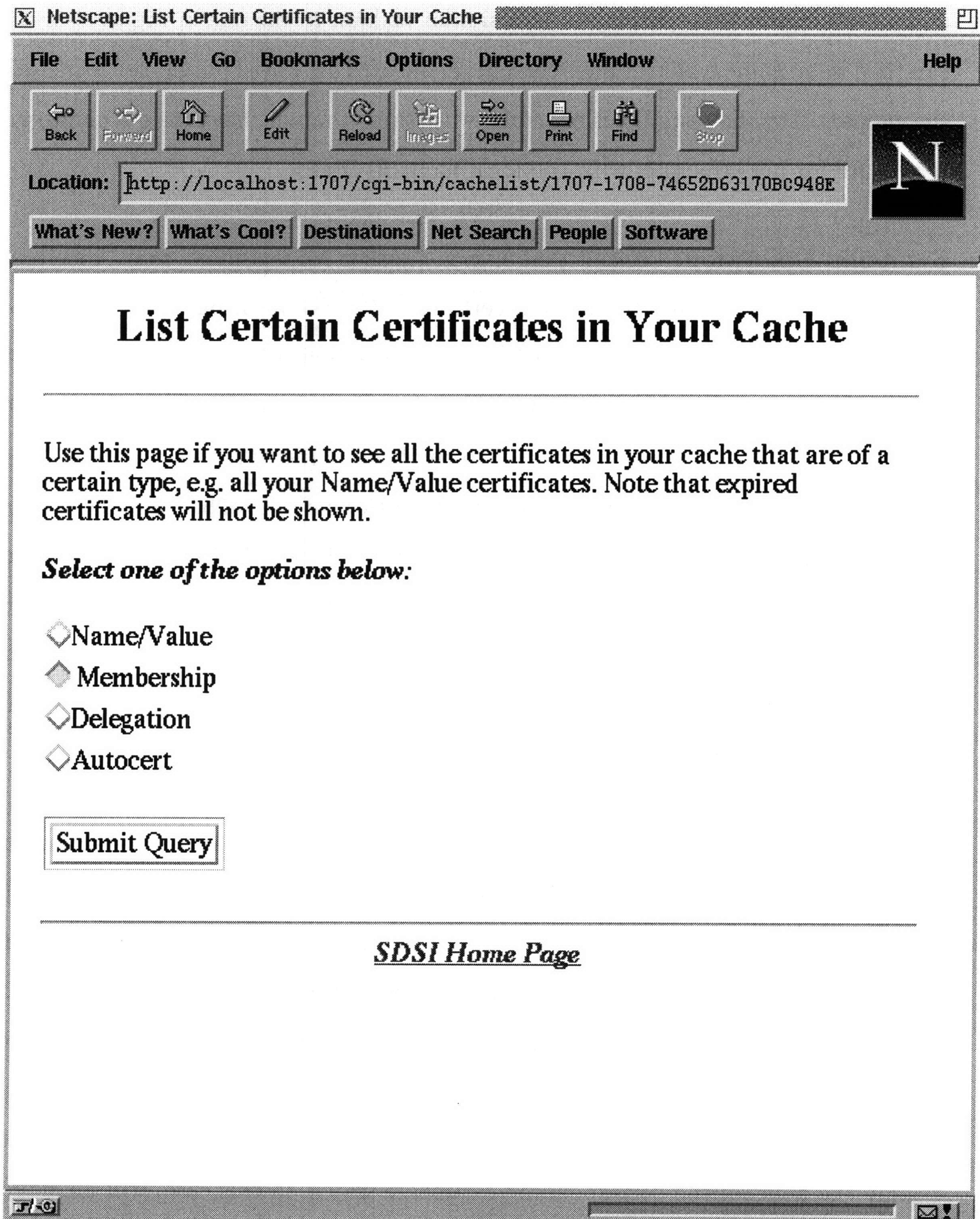

Figure 4-6: List Certificates in Your Cache

Figure 4-7: Result of List Certificates in Your Cache

to place them in a group which she calls `math_lovers`. To do this she goes to the `Define a Group` page from her `SDSI Homepage`. Once she is at this page, she types in the name of the group, and since she knows exactly who she wants to put in the group, she can proceed directly to adding names to it. Figure 4-8 shows the appearance of the screen after Alice has already added Bob and Chris to the group `math_lovers`, and is about to add Eve. She does this by entering the name of Eve's principal and clicking on `Submit Request` while the `Add name to group list` button is checked. Note that the names she enters are the local names that she has bound the public-key principals to: `MyBob, MyChris` and `MyEve`.

When Alice has finished adding members to the group, she can select the `Create the Certificate` button, and click on `Submit Request`. A page will then appear which lists all the members of the group that she has entered, and which will prompt her for an expiration date for the certificate. If she then clicks on the `Sign` button of this page, the group definition certificate will be added to her cache. Figure 4-9 shows the result of signing the `math_lovers` group certificate, with an expiration date of January 12, 1998.

Other. Though these options were not described in detail, similar steps can be taken for Alice to issue delegation certificates and to search her local cache using a template which she specifies.

On almost all of the pages, Alice has the option of going straight back to the `SDSI Homepage` by clicking on the `SDSI Home Page` link. Many pages also contain help page links that the user can click on for more information and instructions.

It should be noted that the SDSI UI does not provide other functionality that may require the use of public-key certificates, such as email authentication or code signing. These are applications that can be built on top of the SDSI library, perhaps in conjunction with the SDSI UI.

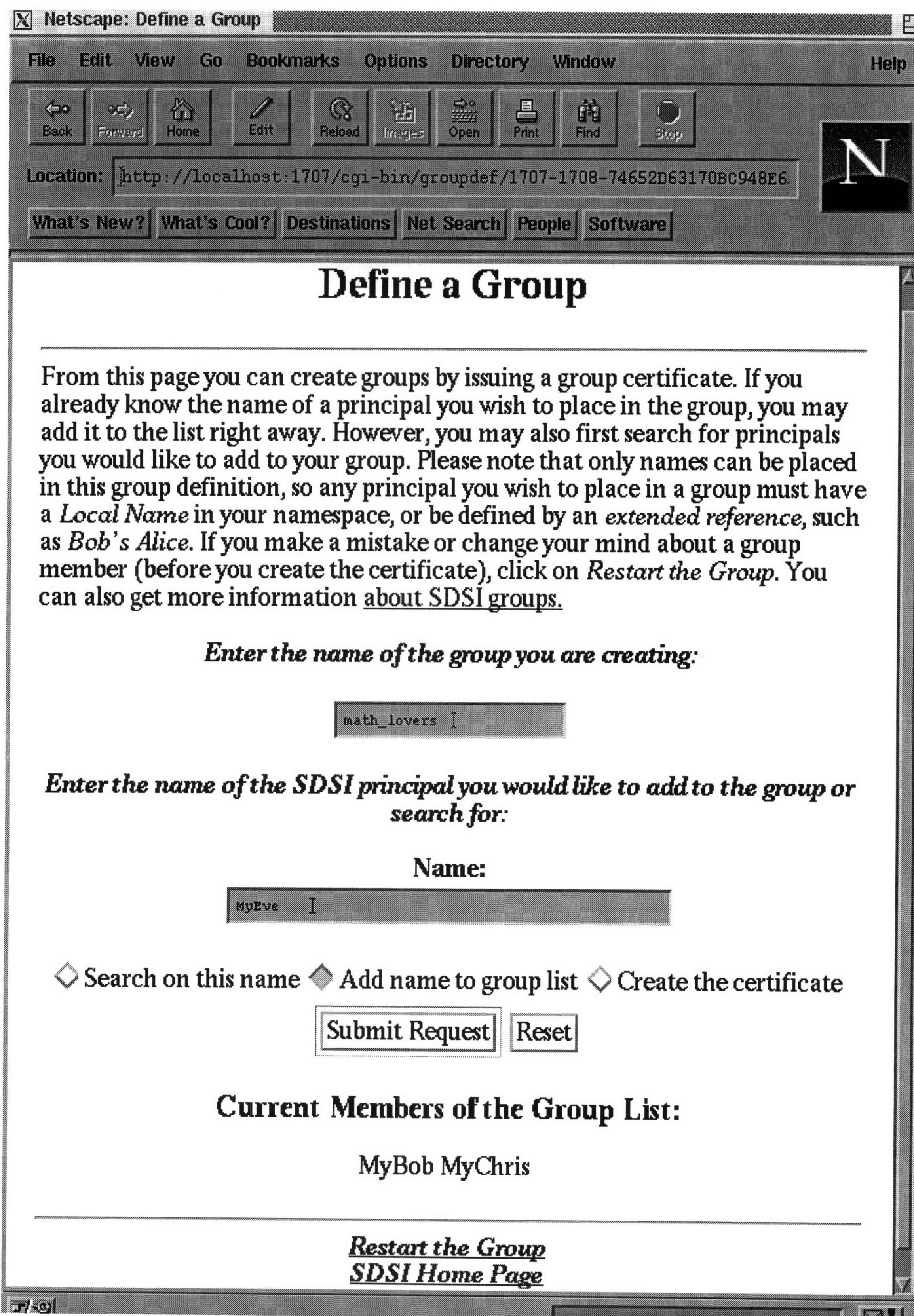

Figure 4-8: Defining a Group

Figure 4-9: Result of Defining a Group

4.4 Design Choices

4.4.1 Structure

When designing the layout of the pages of the SDSI UI, especially the "Homepage", I tried to systematically go through all of the different functions that a user might want to perform, and then arrange them in a suitable way. This sometimes required making assumptions about the potential behavior of future SDSI users, in order to come up with a design that would fit their needs. One of the functions that it is predicted that users will want to perform often is search for SDSI certificates. These certificates can be divided into two main categories:

1. Certificates that are present locally in the user's cache (most of which will probably have been issued by the user)

2. Certificates that are remotely located on other SDSI servers. These certificates can only be obtained by making a query to the appropriate server.

The user will also want to issue certificates for SDSI principals. These could be any of the three main types, *Name/Value, Membership* or *Delegation.* In addition, the user may want to make group definitions (which are in fact just a special form of *Name/Value* certificates).

At certain times, and especially on first using SDSI, the user may obtain a principal from a file, rather than from a query. This may happen if the user obtains the principal from a diskette, for example. In this case, there must be a way to load that principal into the UI.

It is also probable that, at various times, the user will also want to view some of the certificates in his cache, many of which he may have issued himself. There seemed to be three ways of doing this which would be useful:

1. Searching for the principal by local name (probably the most useful way).

2. Searching for all the certificates of a given type e.g. all the membership certificates, or all the name/value certificates.

3. Searching the cache using an arbitrary template, not covered by any of the two options above e.g. searching based on the `Server-At` field of a Principal.

Another operation the user may want to perform is find out whether he is a member of a group that another SDSI user has created. Finally, each SDSI user must have an Autocert, and there needs to be a way for it to be created and updated.

Looking at all of these options together, it seemed as though they could be grouped into two broad categories:

1. Functions for which the user might need to contact other SDSI servers. These included searching for certificates, and making membership queries.

2. Functions which only involved the user's cache and server. These included viewing certificates that were in the cache, and managing the Autocert.

Following this reasoning, the `SDSI Homepage` was divided into the `Search, Issue, and Query` and `Manage Your Local Cache` categories, as shown in Figure 4-1.

It is also predicted that many times a user will want search for and view remote certificates before deciding to issue a certificate for the corresponding principal. For example, Alice may want to see the certificate that Bob has issued for Chris, because some of the information it contains may help her with issuing her certificate. For this reason, the searching and issuing functions were combined together in the `Search for Principals and Issue Certificates` page. It also made sense for the user to be able to search his local cache from this page, using the local name of the principal he is looking for. In addition, since loading a principal or certificate from a file would yield basically the same result as doing a query (that result being a public-key principal or certificate), the option of loading from a file was also placed on the `Search for Principals and Issue Certificates` page.

A similar case holds for when the user wants to create a group. Many times he will want to search (both locally and remotely) for the principals that he wants to add to the group, and then finally make the group definition. This is why the `Define a Group` page, was placed under the `Search, Issue, and Query` section, rather than the `Manage Your Local Cache` section.

At the time this thesis was written, the `Membership Query` function was not fully complete in either the SDSI shell or the UI. However, if it were, it would also have been placed in the `Search, Issue, and Query` category.

4.4.2 Limiting the UI

During the design of the SDSI UI, some decisions were made to limit how much of the underlying functionality the user would be able to view or access. In particular, one of the aims was to hide the user from SDSI's S-expressions as much as possible, and especially to prevent him from having to type in S-expressions, since this could become confusing. As it is, only two pages require the user to type S-expressions, where it is unavoidable. One is the `Search Your Certificate Cache` page on which the user types in a template that will be searched on. The other is the `Issue Delegation Cert` page, which is found in the `Issue Certificates` section of the `Search for Principals and Issue Certificates` page, where the user is required to enter an authorization template for the delegation certificate. Both of these pages will probably only be needed by advanced users of SDSI.

Another design choice was to try to have the user deal only with names, rather than principals. In particular, a user should never have to type a principal into the UI. Even within this limitation, however, there are some potential complications. SDSI has an extremely flexible naming structure, and this raises some issues about how the UI should deal with names. As an example, suppose Alice wants to search for the principal that is defined by the extended reference `Bob's joan`. However, in this case, Bob has bound `joan` to the symbolic reference `Harvard's joan55`, rather than an explicit principal. In turn, Harvard may have defined its `joan55` to be `faculty's joan3`. In this case, from Alice's point of view, there are several ways to refer to joan's public-key principal, which we will denote as `<principal-j>`. These include:

- `Bob's joan`
- `Bob's Harvard's joan55`

- Bob's Harvard's faculty's joan3
- <principal-h>'s joan55
- <principal-h>'s faculty's joan3
- <principal-f>'s joan3

where <principal-h> and <principal-f> are the actual principals of Harvard and Harvard's faculty, respectively. Though each of the definitions listed above should end up with the same principal for joan, that being <principal-j>, the time at which the binding takes place is different in each case, and so they could give different results depending on whether one of the bindings has changed. As an example, if the "<principal-f>'s joan3" definition is used, then the result will change only when <principal-f> changes it's binding for joan3. However, if the "Bob's Harvard's faculty's joan3" definition is used, the result will change when any of these entities changes its bindings. In particular, if the SDSI server of one of these principals is down, joan's principal will be inaccessible to Alice.

If Alice wanted to issue a certificate for joan, a choice of one of these options would have to be made. This could get complicated, and possibly confuse the user, so it might be useful to limit the options somewhat. In the case of this implementation, part of the choice is already made by the underlying library, which evaluates all the way to a principal in every possible case. Therefore, given the extended reference Bob's joan which is defined as described above, the evaluation would not stop until it reached the <principal-j>, which is the only thing that would be returned. However, the user always has the option of issuing a symbolic certificate using a value such as Bob's joan. So in effect, Alice has a choice between using the explicit principal (<principal-j>) in a certificate, or the top-level symbolic definition (Bob's joan). Any of the other extended references could of course be used, but only if Alice found them out by other means.

A similar issue arises with group definitions. Suppose Bob wants to include (the public-keys of) Alice, Eve, and Joan in a group. He could potentially use either a SDSI name or a principal for each of the members. Bob may also want to search for a

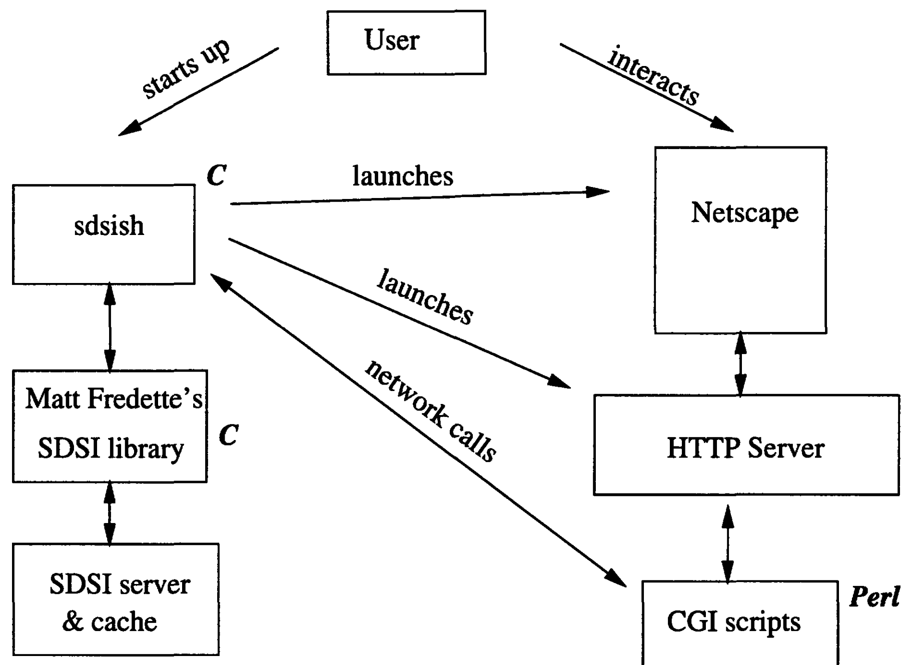

Figure 4-10: The Underlying Architecture

principal before adding it to a group, which once again raises the issue of there perhaps being multiple definitions of a principal. To reduce these complications, the the UI only gives the user the option of entering names in the group definition, whether they are local such as `joan` or extended references, such as `Harvard's joan55`. However, note that actual principals can be indirectly added to a group by issuing membership certificates for them, using the `Search For Principals and Issue Certificates` page.

4.5 The Underlying Architecture

Figure 4-10 illustrates the underlying architecture of the SDSI UI.

When the user types in the command to start up the SDSI UI, a program written in C, called `sdsish` (for SDSI shell) starts up. It prompts the user for a password, and if it is entered successfully, the user's private key (i.e. the private part of the public-key pair) is unlocked. The user's public-key principal definition is installed

into the SDSI library, and that principal becomes the `speaker` of any actions that are performed. `Sdsish` then starts up an HTTP server and launches Netscape (or an appropriate browser). The HTTP server then runs a CGI script which displays the user's `root` SDSI page, the `SDSI Homepage`. From here on the user only interacts with Netscape. Depending on the user's actions, other CGI scripts will be launched, most of which will need to interact with the user's SDSI server and certificate cache. As an example, the user may want to add a newly created certificate to his cache. When this occurs, the appropriate CGI script makes a network call back to `sdsish`. `Sdsish` then makes calls to the SDSI library which in turn interacts with the user's SDSI servers and cache. The results of these calls are passed back to `sdsish`, which in turn returns them to the CGI script that called it, which returns the result to the user, who will view it in Netscape.

The architecture was designed this way in order to allow flexible interaction between the UI and the library. The SDSI shell (`sdsish`) provides higher-level functionality than that of the library: whereas the library operates on C objects and structures, the CGI scripts of the web interface send and receive only text. `Sdsish` therefore acts as an intermediary between the CGI scripts and the SDSI library.

Most of the work of this thesis involved writing the CGI scripts which took the user's queries, passed them to `sdsish`, and then interpreted the response. Some of the main steps involved in this implementation are described in Chapter 5.

Chapter 5

Implementation of the SDSI User Interface

This chapter describes some of the key steps involved in implementing the SDSI User Interface. It assumes some basic knowledge on the reader's part about HTTP servers and CGI scripts. A good overview of these topics can be found in "The HTML Source Book"[6]. All of the code fragments shown in this chapter are written in perl.[1]

5.1 The Web Server

The SDSI UI uses a web browser as the interface for the user to input requests. The requests are submitted using HTML forms, and this requires a web (HTTP) server to be running on the user's machine. The server used in this implementation is NCSA's httpd.

Running a web server on a machine brings up certain security issues. In this case, the user should only be running a local process that accepts connections from the web browser he is currently using to interface to SDSI. Connections should not be accepted from any other locations. This is especially important with the SDSI UI because the server will be launching CGI scripts which will perform functions that

[1]The code for this thesis will be posted at a later date on the SDSI Web Page. [1]

affect the user's SDSI certificates and servers. It would be disastrous if an adversary could gain access to the web server on the user's machine and thus gain access to his SDSI cache, perhaps doing damage or viewing sensitive information.

Luckily, the web server can be configured to accept connections only from the local host i.e. the machine on which the web server itself is running, since this is the machine on which, presumably, the user will also be running the web browser. This adjustment will prevent outside adversaries on remote hosts from connecting to the web server. It does not, however, solve the problem of adversaries that are on the same machine as the user trying to connect to the web server. The possibility of more than one user being logged onto the same machine is fairly large, especially in many of the time-sharing systems that exist today. This problem was solved by using a security loop, as described in section 5.3.

Launching the web server and setting up the configuration files were eventually taken over by the SDSI shell (sdsish) which was written as part of "An Implementation of SDSI"[5] by Matt Fredette. The interaction of the UI and sdsish is described in more detail in section 5.2.

5.2 Interacting with the SDSI Shell (sdsish)

The SDSI shell is the UI's interface to the SDSI library and by extension, the user's servers and cache. It can run in two modes, either in command line mode, which expects typed input from a user, or as a daemon, waiting for network connections from one of the CGI scripts of the UI. The SDSI shell expects the same form of commands in both modes. For example, when defining a variable `a` to be a string, `bob-jones`, sdsish expects the string to be followed by a dot on a line by itself. In command line mode, this would look like:

```
sdsish> define a
bob-jones
.
```

```
sdsish>
```

So, if this same command were to be given in the network mode, it would have to be constructed (in perl) as:

```
$command = ''define a\nbob-jones\n.\nexit\n'';
```

The extra ''exit\n'' at the end is needed when the shell is used in the network mode to signal the end of the connection.

Another important point about sdsish is that it has a predefined variable called the `speaker`, which represents the public-key principal of the user that started up the UI and entered the corresponding password. This is useful because many of the commands of sdsish require the principal that is doing the action to be specified, and this is usually the `speaker`. To make this more concrete, the following is a typical command and response that would occur in the SDSI shell (in command-line mode):

```
sdsish> print speaker
( Principal:
  ( Public-Key:
    ( RSA-with-SHA1:
      ( N:
        #B8D1C7A2EBCE4EDDD346162129A40451999B28AC7747AC9-
        #6E0471936DBCC1EF1684C5769C4441681FC8FE81A6F5003 )
      ( E: #11 ) ) )
  ( Server-At: swan.lcs.mit.edu thrush.lcs.mit.edu ) )
sdsish>
```

Once a command has been set up, a connection has to be made to sdsish. This is accomplished by the `&connect_to_sdsish`[2] procedure, which establishes a filehandle to the SDSI shell, called `SDSISH`. Once this connection has been made, the command itself is flushed to `SDSISH`. The script that did this then waits for the shell's response and interprets it accordingly (this often consists of decomposing a list of

[2] Thanks to Matt Fredette for providing this code.

certificates into an array of individual ones). This is accomplished by a procedure called `&decomplist`.

As an example, suppose the user wants to query his cache for a certificate with the local name `bob`. The correct template for this would be `(Cert: ( Local-Name: bob ))`. The command to be used in sdsish would be of the form `get <principal> <template>` where `<principal>` specifies the public-key that should have signed the resulting certificate(s), and `<template>` is the template that the certificate(s) should match. Putting it all together, to make a query to sdsish, using the template `(Cert: ( Local-Name: bob ))`, the following would be done in one of the CGI scripts:

```
$template = "(Cert: ( Local-Name: bob ))";
$command = "define template\n$template\nget speaker template\nexit\n";
&connect_to_sdsish("SDSISH", $bigcookie);
&printflush(SDSISH, "$command");
($valname, @certarray) = &decomplist(SDSISH);
```

Many of the other function calls to sdsish follow a similar pattern.

5.3 The Security Loop

The security loop is implemented to ensure that only the legitimate user will have access to his SDSI servers and cache during each session that he interacts with the SDSI UI. The idea behind the security loop is simple: each time the SDSI UI is started up, sdsish generates a large random number, and passes it to the web server via the browser. When an access has to be made to sdsish, it expects to receive this number first. If it does not, it will not grant any of the user's requests. This should ensure that only the user that initially started up the UI and sdsish will be able to gain access to that sdsish. If the number generated by sdsish is sufficiently large and random, there is a very low probability that an adversary will be able to guess it, and therefore gain unauthorised access.

To describe the process in more detail, sdsish generates a large random number, referred to hereon as a `cookie`, when the user starts up the SDSI UI (a fresh `cookie` is generated each time the UI is run). The sdsish passes this cookie to the web server as a request variable on the first connection to the server, when the `root` page is started up. Since it is a request variable, the cookie can be received by the `root` CGI script, which is the first page of the UI.

In actual fact, the shell sends not only the `cookie`, but also the port on which the web server (which was launched by sdsish) is running, as well as the port on which sdsish itself is running. The UI scripts need all of this information so that they will know which port of the local host to connect to for the HTTP server, as well as which port to connect to when making calls back to sdsish, and this information will change with each session of use of the UI. The `cookie`, the HTTP server port, and the sdsish port are collectively (joined by hyphens) referred to as the `bigcookie`. So the sdsish would use a URL like `http://localhost:$port/cgi-bin/root?info=$bigcookie` to access the `root` page of the UI.

As this shows, initially, the `root` script receives the `bigcookie`. However, all of the scripts may potentially use the shell or make a connection to the web server, and so they need to know the `cookie`, the HTTP port, and the sdsish port as well. Therefore the `bigcookie` is passed between any two scripts that reference each other. This is accomplished by including it in the URL path of the script that is being referenced. The `bigcookie` then appears as the `PATH_INFO` environment variable, and can be accessed by the referenced CGI script. This script will then split the `bigcookie` into its three components, the `cookie`, the HTTP port, and the sdsish port. The following lines show some of the perl code that makes this happen. Within a referencing script, passing the `bigcookie` will be done by a line such as:

```
<FORM ACTION="http://localhost:$port/cgi-bin/new_script/$bigcookie">
```

Then within the referenced `new_script`, the `bigcookie` will be received and split by the following lines of code:

```
$bigcookie = $ENV{PATH_INFO};
```

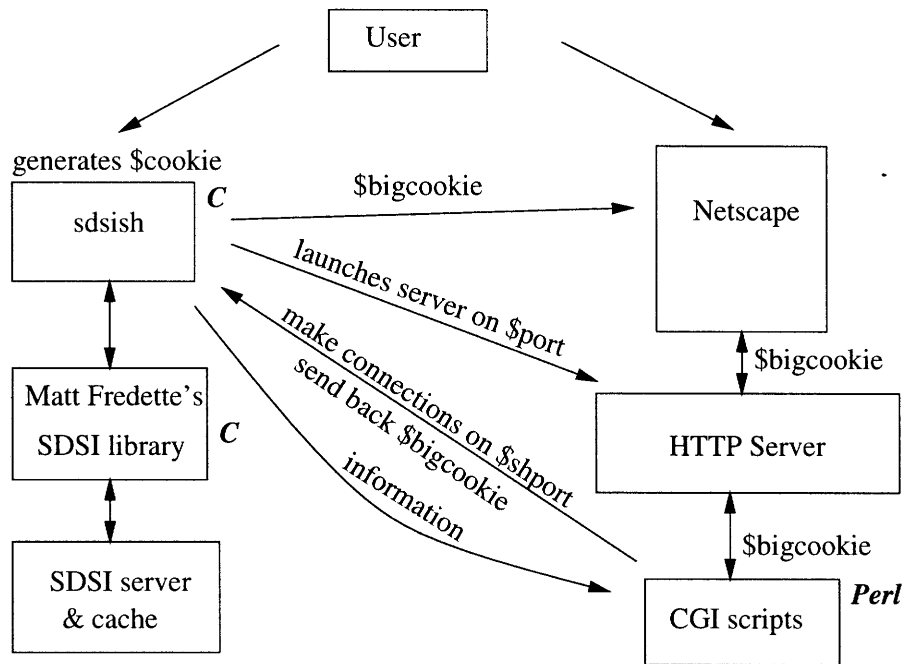

Figure 5-1: The Underlying Architecture with Security Loop

```
$bigcookie =~ s/\/(\w+)/$1/;
($port, $shport, $cookie) = split (/-/, $bigcookie, 3);
```

In this case `$port` represents the HTTP port, `$shport` represents the port of the SDSI shell, and `$cookie` represents the `cookie`, which is a large random number, as described previously.

Finally, when a script wants to make a call back to sdsish so that some function can be performed that accesses the user's cache (for example, a `Get` query), it must first pass the `cookie` to it. In actual fact, the entire `bigcookie` is passed as shown by the line `&connect_to_sdsish("SDSISH", $bigcookie)`, which was also mentioned in Section 5.2. Figure 5-1 illustrates the underlying architecture of the SDSI UI, with the security loop indicated.

In addition, each time a script is accessed by the web server, it checks that some cookie was passed properly, and gives an error if not. This check is not totally necessary, since if a cookie was not passed, then any attempt to use the sdsish would fail, and the sdsish is the only process that actually touches the user's cache, and therefore, could potentially do damage. However, the check provides an extra layer

Once the correct cookie has been passed to the sdsish, the rest of the interaction proceeds as described in section 5.2.

Chapter 6

Discussion and Future Work

This chapter discusses some of the difficulties encountered during the design and implementation of the SDSI UI, as well as some possible future improvements. It also outlines some changes that will need to be made for the next version of SDSI, which is currently in progress.

6.1 Challenges

One of the most challenging aspects of creating the SDSI User Interface was designing it in a way that made it easy for the user. Even after the implementation details of connecting to the SDSI shell and interpreting the response from the library had been worked out, there were still uncertainties about the best way to design the web pages. One thing I learnt from this thesis work is that user interface design is an art in itself, and requires a lot of thought. It is important for the user to always know what state he is in during any given process, and what he can do on each screen that is presented to him. Part of this can be accomplished by having very clear instructions, but designing the flow of the pages in a way that is intuitive for the user is also important.

The challenge of making the UI simple was increased somewhat by SDSI's flexibility. SDSI has a lot of options for the user, and these need to be managed carefully. As an example, after searching for a certificate on the `Search for Principals and`

`Issue Certificates` page as described in Chapter 4, the user has the option of issuing a name/value certificate for the principal that was found, or issuing a membership certificate for it, or issuing any symbolic certificate, or clicking on the help link, etc. It is important to keep the user informed about the whole process, regardless of what choice is made. This can be partly accomplished by carefully and clearly labelling the HTML page titles, instructions and even the `Submit` buttons that the user will click on to send a query to the HTTP server.

In addition, as discussed also in Chapter 4, the SDSI UI is designed with the average user in mind, and this posed a challenge of designing it in a way that did not actually assume too much previous knowledge of any person that would be attempting to use it. Since I knew the SDSI proposal very well by the time I started creating the UI, this was sometimes difficult to do. Many times I had to step back and put myself in the shoes of a person who was only vaguely familiar with SDSI, and try to provide the right amount of support to guide that person through the interface to get at what he or she wanted to do, or even to discover what that was.

6.2 Future Improvements

As described in Section 6.1, designing the SDSI UI was challenging, and future improvements would include making it more user friendly, intuitive, simple, and easy to use. Part of this process will involve extensively testing the user interface on users, to see to see how easily they can adapt to it, and then perhaps making even more design changes.

In addition, there are several ways in which the functionality of the UI could be improved, though time or some other limitation did not permit it in this implementation. Some of these functions are briefly described below.

Running the UI remotely. Currently this implementation assumes that the SDSI shell, HTTP server and web browser, will all run on the same machine. However, users may want the flexibility of running the UI remotely. This might require some

changes in both the SDSI UI and sdsish.

Revoking Certificates from Servers The UI does not currently give users the option to "revoke" certificates from their SDSI servers, though this ability is supported by the underlying library. The result of such a request would be to stop the user's servers from distributing the specified certificate. This is desirable since users should have control over the certificates that their servers distribute.

More Flexible Group Definitions. The UI does not give the user the option to use `AND`, `NOT`, `MINUS`, or `ANY` in group definitions: the members are automatically joined together by `OR`s. However, this default is probably sufficient for much of what a typical user would want to do. Also, when creating a group, the user cannot delete individual members, only the whole group at once. Changes to accommodate these options might be desirable for an implementation of SDSI version 1.0. However, group certificates are changed extensively in the next version of SDSI, which is outlined in Section 6.3.

Images. SDSI certificates may include images, however support for this was not included in either the SDSI shell or the UI.

Standard root support. The SDSI proposal[8] describes certain principals that are likely to become "standard roots". In addition, DNS Internet email names are given special status. Explicit support for this was not included in the UI.

Text fragment searching. Users can currently search for certificates by inputting names, such as `bob-jones`. The UI would be even more useful if it could support searches by text fragments such as `bo`, to find both `bob` and `bob_jones`. Functionality such as this might require changes in both the underlying library and the user interface.

Compatibility with other Schemes. It will be useful and perhaps even necessary for SDSI to have the ability to inter-operate with other public-key certificate formats

such as PGP and X.509. Some work has already been done on converting PGP public-key rings to SDSI principals in "An Implementation of SDSI"[5]. To that end, it would be desirable for the user to have the option of importing a certificate of another type, and perhaps extracting the public key, or converting it into SDSI format.

More Options When Issuing Certs. Currently, users only have the option of entering a local or group name, and an expiration date when they issue certificates (other than the Autocert). SDSI certificates can contain other fields such as `Description`, and the user should be given the option to enter values for them.

Reverse lookup. If a principal is returned from sdsish (for example, as the signer of a certificate obtained from a `Get` query), the actual principal is displayed to the user, rather than a corresponding name. The only exception to this occurs when the principal is actually the user's principal, referred to elsewhere as the `speaker`. Then, instead of the principal, the user is shown the word `YOU`. Given that one of the aims of SDSI is to have users deal with names rather than public-keys, it would be desirable for the local name associated with a principal to be displayed to the user (if one exists), rather than the principal itself. This would require a "reverse lookup", going from a principal to a name in the user's local name space.

ACLs. Each SDSI certificate has an ACL. The UI currently always uses certain default ACLs when certificates are issued by the user. For membership certificates, this default is `(ACL: (member: ALL!))`, and for name/value and auto-certificates, it is `(ACL: (read: ALL! ))` (note that this is actually a departure from what was outlined in the SDSI paper[8]). An improvement would be to give the user the flexibility to specify other ACLs for the certificates that he signs.

Membership Queries. Membership queries were not fully implemented in the user interface or the SDSI shell.

Membership Certs. The UI only allows membership certificates to be issued for actual principals that are obtained via a search, or by loading from a file. It would be desirable for a user to be able to type in just the local or extended name of a principal for which he wishes to issue a membership certificate.

Reconfirmation. Users are not presently given the option to enter a reconfirmation period when they sign certificates.

More Get Query Options. Currently users can only make `Get` queries to other servers with the `Local-Name:` template (which is implied when a search is done by name). They may in fact want to make more complicated requests using templates (like they can in their local caches). In particular, a user should be able to make a request to obtain the Autocert of another user.

GET vs. POST. The UI uses the GET method to submit HTTP requests to the web server. This is extremely convenient for generating queries on the fly and passing information between different CGI scripts. However there is a limit to the amount of data that can be passed using GET (though it is very large). This limit varies from browser to browser. With POST, however, the amount of data that can be submitted is unlimited. Since SDSI certificates can become very large, and may sometimes need to be submitted as the value of a request variable to the server, it may be safer, though less convenient, to switch from GET to POST.

6.3 The Evolution of SDSI

As discussed in previous chapters, SDSI is a relatively new proposal, and it is still in development. In particular, the authors of SDSI and SPKI [3] are actively discussing a merging of the two schemes, since they possess many similarities. This design is called SPKI/SDSI 2.0. It is also referred to as SPKI 2.0 and SDSI 2.0 This thesis implements SDSI version 1.0, and a new version may require modifications of the user

interface. Some of the new features of SDSI 2.0 are described below.

SDSI 2.0 unifies all of the different certificate types present in SDSI 1.0 into one certificate type. Each certificate must contain an `issuer`, a `subject`, and a `tag`, and optionally, a `propagation`, and a `validity`. As in SDSI 1.0, principals can be referred to either as keys or by names.

Tags. Tags are used to give particular authorizations to the subject of the certificate, such as the ability to `telnet` or `ftp` to a particular Internet address. They are what SPKI originally called `auths`. Each certificate has exactly one tag, of the form `(tag T)`, where `T` is an arbitrary S-expression. The tag may also contain various `*-forms` to indicate the authorization that is being given. Some examples of tags are given below:

- `(tag (*))` which represents the set of all S-expressions, and thus denotes unlimited transfer of authorization.
- `(tag (spend-from 1234))` which denotes authorization to spend from account 1234.
- `(tag (spend-from (* set 1234 5678)))` which denotes authorization to spend from either account 1234 or 5678.
- `tag (http (* prefix "http://www.abc.com/"))` which authorizes http access to any web URLs that start with `http://www.abc.com/`.
- `(tag (spend-amount (* range numeric (< 5000))))` authorizes someone to spend up to 5000 dollars.

There are several other proposed `*-forms`.

Propagation. Certificates may have a `propagation` part which specifies the ability of the `subject` of the certificate to pass on the authorization given by the `tag`. If it does exist, the propagation field always appears in the form `(propagation`

`stop-at-key)`. This allows arbitrary propagation of the authority of the certificate through subjects that are names, but stops propagation at the first subject that is a key.

Validity. Certificates may also contain `not-before` and `not-after` fields which specify the validity period of the certificate.

More information about these and other aspects of the design of SDSI 2.0 can be found on the SDSI Web Page [1]. Developments such as these will require corresponding changes in the SDSI user interface.

Chapter 7

Conclusion

This aim of this thesis was to provide a prototype implementation of a graphical user interface for SDSI. Though there are many improvements and additions that could be made, this goal has been achieved, and the SDSI UI can be used to provide most of the basic SDSI features. In retrospect, the web-based approach was a success. Any modifications to the user interface would involve changing the web page design, not the basic underlying architecture. This SDSI UI presents a working prototype, however, designing an interface for an inexperienced user to a relatively complicated system is challenging, and any future work should involve testing the UI from a user's point of view, which may cause the design to go through even more iterations. SDSI itself is going through some design changes, and it is hoped that this thesis work can be used as a basis for future implementations of SDSI.

Bibliography

[1] The SDSI web page. http://theory.lcs.mit.edu/~cis/sdsi.html.

[2] Matt Blaze, Joan Feigenbaum, and Jack Lacy. Decentralized trust management. May 1996.

[3] Carl Ellison, Bill Frantz, and Brian Thomas. Simple public key certificate. June 1996.

[4] Warwick Ford and Michael S. Baum. *Secure Electronic Commerce: Building the Infrastructure for Digital Signatures and Encryption.* Prentice Hall, Inc., 1997.

[5] Matthew H. Fredette. An implementation of SDSI - the simple distributed security infrastructure. Master's thesis, M.I.T, May 1997.

[6] Ian S. Graham. *The HTML Source Book.* John Wiley & Sons, Inc., 1995.

[7] Stephen T. Kent. Internet privacy enhanced mail. *Communications of the ACM*, 36(8):48–60, August 1993.

[8] Ronald Rivest and Butler Lampson. SDSI - a simple distributed security infrastructure. September 1996.

[9] Peter Wayner. Who goes there? *Byte*, 22(6):70–80, June 1997.

[10] Philip Zimmermann. PGP user's guide, volume one: Essential topics. October 1994.