Designing a Distributed Authorization Service*

Thomas Y.C. Woo
Networking Software Research Department
Bell Laboratories
woo@research.bell-labs.com

Simon S. Lam
Department of Computer Sciences
The University of Texas at Austin
lam@cs.utexas.edu

Abstract

We present the design of a distributed authorization service which parallels existing authentication services for distributed systems. Such a service would operate on top of an authentication substrate. There are two distinct ideas underlying our design: (1) The use of a language, called *generalized access control list* (GACL), as a common representation of authorization requirements. (2) The use of authenticated delegation to effect authorization offloading from an end server to an authorization server. We present the syntax and semantics of GACL, and illustrate how it can be used to specify authorization requirements that cannot be easily specified by ordinary ACL. We also describe the protocols in our design.

1 Introduction

Advances in internetworking have transformed distributed systems into a marketplace of *services*. Some of the standard services in today's distributed systems include file service, print service, electronic mail service, and so on. Apart from these "system" services, network users are beginning to offer their own services as well. This is most evident in the rapid growth of WWW, which is essentially a community of "user" services. Typically, these "user" services are more specialized and personal, e.g., financial transactions service.

Security is an important concern in the design and implementation of such services. Design considerations include the following (among others): (1) service is only rendered to authorized clients; (2) proper charges are levied on services performed; and (3) correct records are kept for all services requested and delivered. These considerations give rise to the problems of *authentication*, *authorization*, *accounting*, and *auditing*.

Among these problems, authentication is the most basic, as well as the most studied [17]. On the other hand, the problems of authorization, accounting and auditing have remained relatively unexplored. In this paper, we examine the major issues involved in implementing an authorization service for distributed systems, and propose a specific design that addresses these issues.

Our design is based on two ideas, namely, (1) a language-based approach (called *generalized access control list* or GACL in short) for specifying authorizations; and (2) authenticated delegation. GACL is a significant extension of ordinary ACL. In particular, it provides constructs for explicitly stating inheritance and defaults. The expressiveness of GACL allows authorization requirements to be succinctly and uniformly specified.

Authenticated delegation allows a server to securely delegate its authorization functions to specialized authorization servers. The concept of authenticated delegation is not new. For example, it has been discussed in one form or another in [5, 7, 10]. However, most of these works, with the notable exception of [10], concentrate on the authentication aspect. Our study of authenticated delegation is for authorization purposes, and is similar to the notion of *proxy* in [10].

Our goal is to construct an authorization service which parallels existing authentication services. Since our focus is on authorization, we will discuss accounting and auditing issues only to the extent that they are relevant to authorization.

The balance of this paper is organized as follows. In Section 2, we motivate and identify the major issues in designing an authorization service. In Section 3, we informally describe the architecture of our design and the operation of various protocols.[1] In Section 4, high-level specifications of the protocols are presented. In particular, we describe the use of authenticated delegation in our design. In Section 5, we present the formal syntax and semantics of the GACL language. The semantics we use is procedural in nature and hence is fairly close to implementation. In Section 6, we compare our approach to related proposals. Section 7 has some concluding remarks.

2 Motivation

To obtain a service, a client may need to contact a number of servers. For example, to obtain a file from a file service, a client may need to first contact an authentication server to obtain the necessary credentials. In the following, we will refer to a service that a client would ultimately like to obtain as an *end service*; and a server implementing such a service as an *end server*.[2]

*Research sponsored in part by NSA INFOSEC University Research Program under contract no. MDA 904-94-C-6106 and by the Texas Advanced Research Program grant no. 003658-063.

[1] A preliminary overview of our architecture and protocols, together with an informal introduction to the GACL language, have been presented in an extended abstract [16].

[2] This terminology is adapted from [10], where the notion of an end server is defined in the context of a proxy, and is much more specific. Our notion of an end server is informal, and is intended mainly for differentiating user-oriented services from system-oriented services.

```
    service S
(1)     chan := register(S, Service − Locator, ...);
(2)     while msg = listen(chan, ...) do
(3)         id := getid(msg, ...);
(4)         req := getreq(msg, ...);
(5)         if authorized(id, req, errcode, ...)
(6)             do-operation(id, req, ...);
(7)             update-account(id, req, ...);
            else
(8)             reply(chan, errcode, ...);
            end;
(9)         log(id, req, ...);
        end;
    endservice S;
```

Figure 1: Structure of a Typical Service

2.1 End Services

To better understand the issues we address in this paper, we begin by examining the structure of a typical service in a distributed system (see Figure 1).[3]

The structure should be self-explanatory. We provide just a brief description here. In line (1), the server begins by registering itself with a well-known service locator to announce its availability to potential clients. A service locator can be a name server or a registry of remote procedures. The function $register$ returns a handle $chan$ for a channel on which the server should listen for client requests. In line (2), the function $listen$ returns the next message arriving from $chan$. But if no message is available, it blocks until a message arrives. Each message represents a client request. The specific format of a client request is application specific. But it should contain information on the identity of a client, the service desired and possibly other information needed by the server to provide the service.

The function $getid$ in line (3) returns the identity of the client in msg, and the function $getreq$ in line (4) returns the service requested by the client. The return value req typically specifies an operation to be performed, an object on which the operation should be performed, and possibly a list of arguments providing information for carrying out the operation. The types of objects and the operations allowed are application specific. Using such information, authorization is determined by calling the function $authorized$ in line (5). If $authorized$ returns true (indicating grant), the server proceeds to satisfy the request by calling function *do-operation* in line (6). Accounting is performed by calling function *update-account* in line (7). If $authorized$ returns false (indicating denial), a notice is sent to the client in line (8). Lastly, logging is done in line (9) by function log.

2.2 Core Services

The functions $getid$, $authorized$, *update-account* and log implement solutions to the four problems discussed in Introduction. In most existing systems, each end server performs its own authentication, authorization, accounting and logging.

A better approach would be to factor these functions out and implement them separately as a set of *core services* that can in turn be used as a basis for building other (core or end) services. In other words, we want to offload as many common functionalities as possible from user-oriented end servers to system-oriented core servers. Clearly, the success of this approach depends heavily upon whether these functions are generic across different applications.

Among these functions, $getid$ is the most generic. Specifically, there exist notions of identity[4] that are applicable to most services. Indeed, much success has been achieved in abstracting $getid$ and isolating it as a separate authentication service (e.g., [4]). There is even a proposal to standardize an *application program interface* for authentication services [8].

Progress on abstracting the other functions has been much slower. This may be attributed to the perception that these functions are not as generic. For example, authorization is often perceived to be tightly coupled to an application and hence cannot be easily abstracted.

2.3 Authorization Service

Our research aims at abstracting and separating out the function $authorized$ as a core service, which performs authorization on behalf of end servers. A client desiring service from an end server must first contact an authorization server (and possibly an authentication server before that) to obtain authorization.

A separate authorization service offers many advantages: (1) Savings in re-implementation effort for each end server. (2) End servers are relieved of the task of determining authorization, which can lead to higher throughput. (3) A specialized authorization service can afford the use of better methods in determining authorization than would be justified for individual end servers. (4) An authorization service can be verified to be secure once and for all, which would make it easier to verify the security of an end service. (5) Anonymity (if desired) can be achieved with the use of a trusted authorization service; see Section 3.2. (6) A uniform authorization service can contribute to uniformity of accounting and auditing functions, hence facilitating the construction of specialized accounting and auditing services.

Two key problems need to be addressed in constructing an authorization service.

Representation. Commonalities in authorization requirements of end servers should be identified, and a representation abstraction designed to capture these commonalities. In our research, we adopt a language approach. Our specification language GACL can be used to specify most commonly encountered authorization requirements, and efficient algorithms can be constructed for their evaluation.

Protocol design. Secure protocols are needed for offloading authorization from end servers to authorization servers, and for interactions among clients, authorization servers and end

[3] For simplicity, we consider an *iterative* server. A *parallel* server has a similar structure.

[4] Some examples are GIDs (global unique ids) and domain names.

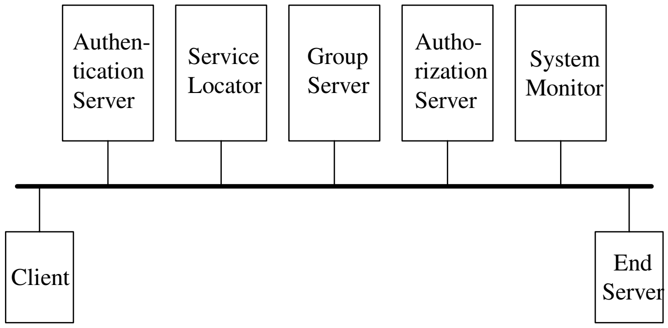

Figure 2: Architecture of Authorization Service

servers. These protocols are intended to make transparent the decoupling of authorization from end services.

In Sections 4 and 5 respectively, we describe in detail how the above two problems are addressed in our design. But first, we provide an informal overview in the next section.

3 Overview of Design

3.1 Architecture

Figure 2 shows the architecture of our design. Below, we give a functional description of the servers shown in the figure. An operational description of the authorization service is provided in Section 3.2.

Service Locator. A service locator assists clients in locating servers implementing a particular service. A service locator obtains such information either statically from some configuration file or dynamically from registration messages sent out by active servers. A service locator functions in a manner similar to a name server[5] or a remote procedures registry. It responds to a client's request with a list of end servers that implement the requested service, and possibly also a list of authorization servers for the end servers (for those that have elected to offload their authorization functions).

Authentication Server. An authentication server performs two basic functions: (1) To authenticate users during their initial sign-on and supply them with an initial set of credentials. (2) To enable mutual authentication between clients and servers. We note that all communications should be authenticated, including those between clients and servers (e.g., clients and group servers, clients and authorization servers), and those between servers (e.g., end servers and authorization servers, system monitors and authorization servers).

Authorization Server. An end server can elect to offload its authorization function to an authorization server. To do so, it needs to *contract* an available authorization server by means of a *contracting protocol*. We will say more about this protocol in the next subsection. An authorization server hands out

[5]Indeed, it can be easily implemented as part of an existing name server mechanism (e.g., DNS) by including additional forms of *resource records*.

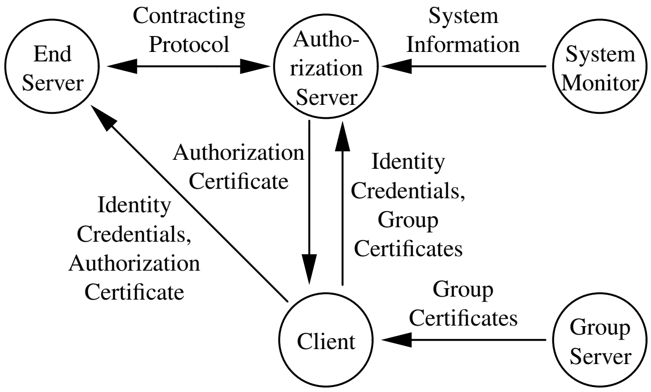

Figure 3: Message Exchanges in our Design

authorization certificates to authorized clients. These certificates are to be forwarded by clients to end servers along with their requests.

Group Server. A group server maintains and provides group membership information. From the perspective of authorization, its main function is to hand out two types of certificates: *membership* and *nonmembership* certificates. The former asserts that a client belongs to a particular group while the latter asserts the opposite. These certificates are requested by clients, and are to be forwarded by clients to an authorization server together with their requests.

System Monitor. A system monitor tracks the values of system predicates. Typically, this is done by the monitor as well as a set of processes executing a distributed algorithm. Such a system monitor, however, cannot be expected to return the precise value of a system predicate at a particular time due to the asynchronous nature of distributed computation. Rather, if a system predicate is *stable*, then the monitor would eventually return its correct value.

These servers are assumed to be *trusted*. For example, a group server is trusted to maintain and hand out correct membership information. A standard technique to ensure such trustworthiness is to implement these servers on dedicated machines that are physically secure.

3.2 Operation

In this section, we describe the operational aspects of our design. Due to space limitation, we will discuss just the key ideas and omit details such as message format, file format and encryption/decryption issues. Our ideas are also illustrated in Figure 3 where, for clarity, we have omitted exchanges that involve the authentication server and service locator. High-level specifications of protocols discussed here are provided in Section 4.

When an end server E, which has elected to offload its authorization, starts up, it locates an authorization server, say A, and uses a contracting protocol to contract A for authorization services. The protocol performs several functions:

- It mutually authenticates E and A, and distributes a new secret session key k for use between E and A.
- It establishes a *delegation key* k_d between E and A. The key k_d will be used by A to sign authorization certificates.
- It transfers an authorization specification *spec* from E to A, which contains authorization requirements specified in GACL, and will be used by A to determine authorization. The integrity of *spec* is protected by signing it with the session key k.[6]

Upon successful contracting, E notifies the service locator that A is its authorization server. Subsequently, the service locator directs clients of E to A to obtain the proper authorization first.[7]

An authorization is typically in the form of an *authorization certificate* signed by A using k_d. The certificate contains, among other information, an *authorization key* k_a that is only known to C (and A of course). C can later submit this certificate to E to obtain its desired service. Knowledge of k_a is used by C to demonstrate to E that the authorization certificate was indeed obtained from A. This method is what we call *authenticated delegation*. A similar method but with the name *proxy* is used in [10]. See Section 6 for a comparison of the two methods.

Server A issues the appropriate authorization certificate to C only after it has determined from *spec* that C can be granted access to E. The determination procedure may require C to submit certain group certificates to satisfy A, and can be iterative. That is, as A examines the entries in a particular generalized access control list, it may request that C furnish additional group certificates.[8] Indeed, C may not be aware of the group certificates required until instructed by A; this is typically the case when nonmembership certificates are needed by A. Hence, several message exchanges may be necessary before an authorization can be determined.

Clearly, caching could be done to enhance efficiency. Caching and the related issue of certificate expiration have correctness implications. For example, if cached group certificates are not invalidated when group membership changes, there may be incorrect grant or denial. Similarly, an unexpired authorization certificate should be invalidated when the particular authorization has been revoked. These issues are similar to those in the use of *capabilities* [6], and are beyond the scope of this paper.

[6]This is similar to a *zone transfer* in DNS, except that authorization data are involved here.

[7]Such redirection is similar to the use of MX records for *mail exchanges* in DNS. A major difference is that mail exchanges are responsible for forwarding mail to their final destinations, while authorization servers do not forward their decisions directly to end servers.

[8]This is commonly known as the *push* model. A *pull* model is one in which A itself gathers the relevant certificates from the group servers. However, it appears to be more desirable to reduce the load of A so that it does not become a bottleneck, even at the expense of the clients.

4 Protocols

We present high-level specifications of protocols introduced in the previous section. These specifications serve to convey key ideas behind the protocols without restricting implementation flexibility. However, due to space limitation, we will omit discussion on practical details.

4.1 Mutual Authentication

Our authorization service assumes the availability of an underlying authentication substrate. This authentication substrate provides two basic functions: (1) authentication of users at initial sign on, and (2) mutual authentication between processes. The initial sign-on returns a set of initial credentials to a user. For example, the set of initial credentials typically includes a certificate signed by an authentication server that contains the user id, the network address of the sign-on host, a timestamp and a lifetime. All clients invoked by the user would inherit this initial set of credentials.

We prefer to use a public key based system.[9] Since the initial sign-on procedure is dependent on specific hardware configurations, we present only our client-server mutual authentication protocol here.

The following shows the basic client-server mutual authentication protocol from [14, 17], where C denotes a client, E a server, and S an authentication server. Also, k_X and k_X^{-1} denote respectively the public and private keys of X, and k the new secret session key to be used between C and E.

(1)	C	:	generate new nonce n_C
(2)	$C \rightarrow E$	:	C, n_C
(3)	C	:	generate new nonce n_E
(4)	$E \rightarrow S$	:	C, E, n_C, n_E
(5)	S	:	generate new session key k
(6)	$S \rightarrow E$	:	$\{\{n_C, n_E, C, E, k\}_{k_S^{-1}}\}_{k_E}$
(7)	$E \rightarrow C$	:	$\{\{n_C, n_E, C, E, k\}_{k_S^{-1}}\}_{k_C}, \{n_C, n_E\}_k$
(8)	$C \rightarrow E$	:	$\{n_E\}_k$

Authentication technology is still evolving. The choice of which authentication protocol to use depends on many factors. We have structured our design in a modular way. Thus, any mutual authentication protocol that provides an authenticated, integrity-protected, secret channel would suffice. Indeed, any of the existing authentication systems could have been used (e.g., [4, 9]).

4.2 Authenticated Delegation

The basic idea of an authenticated delegation is fairly straightforward. Consider two processes P and Q. After mutual authentication, as specified above, P and Q share a secret channel k.[10] If P wants to delegate to Q, it can generate a new secret key k_d and send it to Q via channel k. Since channel k is integrity-protected and secret, only Q can receive k_d.

[9]This preference is mainly due to the *broadcast* ability of public key, which allows easier integration of diverse servers (e.g., group servers, authentication servers and authorization servers).

[10]For convenience, we use the session key distributed in the mutual authentication to refer to the channel.

Contracting:

(1)	$k = \textit{mutual-authenticate}\ (E, A)$		
(2)	E	:	generate a delegation key k_d
(3)	$E \xrightarrow{k} A$	:	$k_d, spec$

Authorization:

(1)	$k' = \textit{mutual-authenticate}\ (C, A)$		
(2)	A	:	generate an authorization key k_a
(3)	A	:	compute $cert = \{k_a, T, L, \textit{other-info}\}_{k_d}$
(4)	$A \xrightarrow{k'} C$	:	$cert, k_a$

End Server Request:

(1)	$C \rightarrow E$	:	$cert, \{T'\}_{k_a}$

Figure 4: Use of Authenticated Delegation

Thus, any message subsequently received by P that has been encrypted by k_d must have come from Q, and can be accepted by P as according to the delegation.

Indeed, Q can further delegate to another process R by generating a new delegation key k_a and providing to R both k_a and a *delegation certificate* of the form

$$cert = \{k_a, T, L, \textit{other-info}\}_{k_d}$$

where T is a timestamp and L a lifetime. If $cert$ is presented to P, P can verify, by the encryption k_d, that $cert$ has been issued by its delegate Q. And R can prove that it is the legitimate "owner" of $cert$ by demonstrating its knowledge of k_a using an *authenticator* of the form $\{T'\}_{k_a}$, where T' is a timestamp.

The use of authenticated delegation in our design is illustrated in Figure 4, where end server E, authorization server A and client C correspond to P, Q and R, respectively, in the above discussion. The notation $k = \textit{mutual-authenticate}(P, Q)$ specifies the execution of a mutual authentication protocol between P and Q that distributes a session key k. A step of the form $P \xrightarrow{k} Q : M$ specifies that message M is sent by P to Q via channel k. Also, in the context of an authorization service, the delegation key k_a and its delegation certificate, described above, are called authorization key and authorization certificate, respectively.

5 The GACL Language

Terminology. To differentiate between our language of generalized access control list from a particular generalized access control list, we will refer to the former as GACL and the latter as gacl. A similar convention (ACL and acl) is adopted in referring to ordinary access control lists. □

Various implementations of ACL have long been used for specifying authorization requirements. An acl is typically associated with an object and consists of a list of pairs of the form (s, R) where s a subject identifier and R a set of access rights. A subject s is granted access r to object o if and only if the acl associated with o contains a pair (s, R) such that $r \in R$.

```
< SimplSubj >   :=  ["−"]< SubjID >
< CmpdSubj >    :=  < SimplSubj > {" ∧ " < SimplSubj >}
< SubjExpr >    :=  ["−"]"∗" | < SimplSubj > | < CmpdSubj >
< SubjList >    :=  < SubjExpr > {"," < SubjExpr >}
< OpExpr >      :=  ["−"]"∗" | ["−"]< OpID >
< OpList >      :=  < OpExpr > {"," < OpExpr >}
< SubjOpPair >  :=  "⟨[" < SubjList > "], [" < OpList > "]⟩"
< STriple >     :=  < ObjID > "::" < SubjOpPair >
< DTriple >     :=  "default::" < SubjOpPair >
< ITriple >     :=  ("always" | "demand") "inherit" < STriple >
< EntryHead >   :=  (< STriple > | < Pred >) {" ∧ " < EntryHead >}
< EntryBody >   :=  < STriple > | < DTriple > | < ITriple >
< GACLEntry >   :=  [< EntryHead > " ⇒ "] < EntryBody >
< DCL >         :=  {( ordered | anonymous)}
< GList >       :=  < GACLEntry > {"," < GACLEntry >}
< gacl >        :=  < ObjID >   "declare" < DCL >
                                "list" < GList >
```

Figure 5: Grammar for GACL

Denial is implicit, i.e., it is implied by the absence of positive authorization in the list.

The key advantage of ACL is its straightforward semantics. However, it is not very expressive. Several extensions have been proposed, e.g., allowing explicit *negative* authorizations. Most of these extensions are, however, ad-hoc and have often been introduced without a well-defined semantics.

We believe that ACL is the right abstraction to use in an authorization service. However, it must be extended to be effective. To this end, we propose the GACL language. GACL is much more expressive than ordinary ACL. The main features of GACL include the following:

- It provides constructs that can express in a straightforward way most commonly encountered authorization requirements. For example, the structural properties, *closure*, *inheritance* and *defaults*, identified in [15], can be directly expressed in GACL.

- It allows *incomplete* authorization to be specified. That is, it is possible that for some request, neither grant nor denial can be determined. A *failure* is returned in this case. This is preferred over the "denial by default" style of authorization because a failure may suggest an error in a specification.

- It has an implementation independent semantics, thus allowing implementations of varied complexity and permitting interoperability across different authorization servers.

- It provides a declaration section that gives an authorization administrator additional flexibility in expressing authorization requirements.

GACL can be viewed as a practical "approximation" of the logical language of *policy base* introduced in [15]. We defer a comparison of the two to Section 6. In the following, we provide a rigorous specification of the syntax and semantics of the GACL langugage. An example specification using GACL is presented in Section 5.4.

5.1 Syntax

We begin with a set ObjID of *object identifiers*, a set SubjID of *subject identifiers*, a set OpID of *operation identifiers*, and a set Pred of *predicate symbols*. We assume that SubjID is partitioned further into two disjoint sets IndID and GroupID. The set IndID contains names for individuals while the set GroupID contains names for groups. An object identifier names an object, e.g., a file, a printer. An operation identifier names an operation, e.g., read, write. Note that not all operations are meaningful on all objects. A predicate symbol denotes a condition on the system and is possibly monitored by some system monitor.

Each object is uniquely associated with a gacl. For that reason, we simply use an ObjID to refer to a gacl in the following. A grammar for the syntax of a gacl is given in Figure 5. We try to keep the grammar as simple as possible. In particular, we have not encoded a number of syntactic restrictions into the grammar to avoid complicating it. They are listed in the following instead. The motivations for these restrictions will become clear when we present the semantics of GACL in the next subsection.

Terminology. The ObjID in a STriple is called an *object modifier*. Given a STriple t, we denote its object modifier by $objm(t)$ and its SubjOpPair part by $pair(t)$. Two OpExpr's are *complementary* if one is the negation of the other, e.g., R and $-\mathsf{R}$ are complementary. Given an OpExpr *op*, we denote the OpExpr complementary to *op* by $\overline{op}$. Let e be a GACLEntry, we denote by $head(e)$ and $body(e)$ respectively the EntryHead part (if present) and the EntryBody part of e. Given a gacl obj, we will refer to its declaration (i.e., DCL part) as obj.**declare** and its list (i.e., GList) part as obj.**list**. □

Here are the restrictions on the grammar:

- A particular SubjID can occur at most once in a CmpdSubj. Specifically, expressions like $\mathsf{Alice} \wedge \mathsf{Alice}$ and $\mathsf{Alice} \wedge -\mathsf{Alice}$ are ruled out.
- An OpList cannot contains a pair of complementary OpID's.
- If "$*$" occurs in a SubjList or an OpList, then it must be the only expression. In addition, a "$*$" in OpList should be understood as an abbreviation for the list of all OpID's. That is, if OpID $= \{\mathsf{R}, \mathsf{W}\}$, then an occurrence of "$*$" in an OpList stands for R,W while an occurrence of "$-*$" stands for $-\mathsf{R}$, $-\mathsf{W}$.
- A gacl cannot inherit from itself. That is, the object modifier in an ITriple must name an ObjID different from the object associated with the gacl in which the ITriple occurs.
- All object modifiers occurring in an EntryBody outside of an ITriple must name the object associated with the current gacl, and hence can be omitted by convention.
- If "ordered" is declared in a gacl, then the modifiers **always** and **demand** in front of **inherit** can be omitted.

We allow a shorthand notation using variables. A GACLEntry containing variables is interpreted as standing for the set of GACLEntries obtained by instantiating the variables with all possible terms of compatible type. We note that the instantiation is performed only on individuals. A instantiation including group identifiers (hence allowing negation as well) could have been defined, but we choose to keep the semantics simple at this point.

Also, in the grammar, the EntryHead part of a GACLEntry is optional. But to simplify the presentation of semantics in the next subsection, it would be useful to assume that the EntryHead is always present. Specifically, we adopt the following convention: The EntryHead of a GACLEntry is always understood to contain a fixed conjunct T, where T is a new distinguished predicate symbol in Pred. That is, if e is a GACLEntry without an EntryHead, it should be interpreted instead as $\mathsf{T} \Rightarrow e$; and for a GACLEntry e that has an nonempty EntryHead, it should be interpreted as $head(e) \wedge \mathsf{T} \Rightarrow body(e)$.

An *authorization specification* is a finite set of gacl's.

We note that the GACL language allows direct expression of *closure*, *default* and *inheritance* properties [15] via the use of its "$\Rightarrow$", **default** and **inherit** constructs respectively.

5.2 Semantics

A gacl specifies a set of authorizations. To define a semantics for GACL, we need to define what is an authorization and how to construct the set of authorizations specified by a gacl.

We first introduce a set *Ind* of individuals; this set is the semantic counterpart of the set IndID. Each element of IndID names a unique individual in *Ind*, and each individual in *Ind* is named by some identifier in IndID. Similarly, we define a set $Op = \{r^+, r^- \mid r \in \text{OpID}\}$ as the semantic counterpart of OpID. Specifically, if $r \in$ OpID, then r names r^+ and $-r$ names r^-. In the following, an OpExpr and the name it stands is used interchangeably. As is the case with OpID's, we say r^+ and r^- are *complementary*, and define $r^+ = \overline{r^-}$ and vice versa. A *compound individual* is an expression of the form $i_1 \wedge \ldots \wedge i_n$ ($n \geq 1$) where each i_k belongs to *Ind*. The operator $\wedge$ is assumed to be commutative, associative and idempotent. In other words, $\mathsf{a} \wedge \mathsf{b}$ denotes the same expression as $\mathsf{b} \wedge \mathsf{a}$, so do $(\mathsf{a} \wedge \mathsf{b}) \wedge \mathsf{c}$ and $\mathsf{a} \wedge (\mathsf{b} \wedge \mathsf{c})$, and $\mathsf{a} \wedge \mathsf{a}$ and a.[11] Given *Ind*, we denote the set of all compound individuals by *CInd*. We note that *Ind* $\subseteq$ *CInd*.

We assume that the following mappings are given: (1) *grpmember* : GroupID $\mapsto$ *PowerSet*(*Ind*). This gives the membership of each group. (2) Φ : Pred $\mapsto \{\mathsf{true}, \mathsf{false}\}$ with the property that $\Phi(\mathsf{T}) = \mathsf{true}$. This gives the current system state.

An *authorization* is an expression of the form $\langle [S], [op] \rangle$, where $S \subseteq$ *CInd* and *op* $\in$ *Op*. For example, $\langle [\{\mathsf{a} \wedge \mathsf{b}, \mathsf{c}\}], [\mathsf{R}^+] \rangle$ is an authorization. This says that both the compound individual $\mathsf{a} \wedge \mathsf{b}$ and the individual c are allowed to perform

[11] Indeed, in abstraction, we can understand a compound individual $i_1 \wedge \ldots \wedge i_n$ as the set $\{i_1, \ldots, i_n\}$.

function *members* (s : SubjExpr) : **set of** $CInd$ {
  case s **of**
    Individual: **return** $\{s\}$;
    $*$: **return** $CInd$;
    $-*$: **return** $\emptyset$;
    default: **let** $s = G_1 \wedge \ldots \wedge G_\ell \wedge -G_{\ell+1} \wedge \ldots \wedge -G_M$
      return
      $$\left\{ \begin{array}{l} i_1 \wedge \ldots \wedge i_\ell \wedge i_{\ell+1} \wedge \ldots \wedge i_n \in CInd \mid \\ \forall\, 1 \leq u \leq \ell : i_u \in G_u \text{ and} \\ \forall\, \ell+1 \leq u \leq m : \nexists\, 1 \leq v \leq n : i_v \in G_u \end{array} \right\}$$
}

function *expand* (p : SubjOpPair) : **set of** Authorization {
  declare S : **set of** SubjOpPair; A : **set of** Authorization
  let $p = \langle [subjlist], [oplist] \rangle$;
  $S := \{\langle [s], [op] \rangle \mid s \in subjlist, op \in oplist\}$;
  $A := \{\langle [\emptyset], [op] \rangle \mid op \in OP\}$;
  for each $\langle [s], [op] \rangle \in S$ **do**
    $A := (A - \{\langle [R], [op] \rangle \mid \langle [R], [op] \rangle \in A\})$
      $\cup \{\langle [members(s) \cup R], [op] \rangle \mid \langle [R], [op] \rangle \in A\}$;
  end;
  return $\{\langle [X], [y] \rangle \in A \mid X \neq \emptyset\}$;
}

function *simplify* (obj : ObjID; l : GList) : GList {
  declare ℓ : GList
  loop each $e \in l$ **do**
    if e is disabled
      next;
    remove from $head$ all $p \neq \mathsf{T} \in$ Pred;
    remove from $head$ all $t \in$ STriple such that $obj \neq objm(t)$;
    $\ell := \ell, head \Rightarrow body$;
  end;
  return ℓ;
}

Figure 6: Some Basic Functions

operation R on the object that is associated with this authorization. We say authorizations $\langle [S], [op] \rangle$ and $\langle [S'], [op'] \rangle$ are *contradictory* if $S \cap S' \neq \emptyset$ and *op* and *op'* are complementary. A set of authorizations is *inconsistent* if it contains contradictory authorizations. Whether a gacl defines a consistent set of authorizations depends on the mappings *grpmember* and Φ, and hence cannot be determined statically in general.

A gacl is *ordered* if "ordered" is in its declaration; otherwise it is *unordered*. The semantics of ordered gacl and unordered gacl are defined differently, and are presented separately below. The semantics to be presented is *procedural* in nature. It is simpler, more intuitive and closer to implementation than a *declarative* semantic, though a declarative (and possibly more complicated) one could have been given by translating the GACL language to a logical language. To sum up, given a gacl *obj*, we define:

$$\textit{semantics}\,(\textit{obj}) = \left\{ \begin{array}{ll} \textit{OrderedGACL}\,(\textit{obj}) & \text{if ordered} \in \textit{obj}.\textbf{declare} \\ \textit{UnorderedGACL}\,(\textit{obj}) & \text{otherwise} \end{array} \right.$$

where definitions of the procedures *OrderedGACL* and *UnorderedGACL* are given below.

We next introduce a few functions that are needed later in our definition of semantics. These functions are shown in Figure 6. The function *members*(), when given a SubjExpr s returns a set of compound individuals denoted by s. The function *expand*() takes a STriple t and returns the set of authorizations specified by t. We provide here some example applications of *members*() and *expand*(). Suppose $\mathsf{o} \in$ ObjID, $\mathsf{G} \in$ GroupID, $\mathsf{R} \in$ OpID, Ind $= \{\mathsf{a}, \mathsf{b}, \mathsf{c}\}$ and *grpmember*$(\mathsf{G}) = \{\mathsf{b}, \mathsf{c}\}$, Then *members*$(\mathsf{a} \wedge \mathsf{G}) = \{\mathsf{a} \wedge \mathsf{b}, \mathsf{a} \wedge \mathsf{c}\}$ and *expand*$(\langle [\mathsf{a}, \mathsf{a} \wedge \mathsf{G}], [-\mathsf{R}] \rangle) = \{\langle [\{\mathsf{a}, \mathsf{a} \wedge \mathsf{b}, \mathsf{a} \wedge \mathsf{c}\}], [\mathsf{R}^-] \rangle\}$.

Let A be a set of authorizations. Let $p \in$ SubjOpPair. We say A *satisfies* p if *expand*$(p) \subseteq A$.

Given a GACLEntry e in a gacl *obj*, we say e is *disabled* if there exists a Pred $p \in head(e)$ such that $\Phi(p) = \mathsf{false}$ or there exists a STriple $t \in head(e)$ such that $obj \neq objm(t)$ and *semantics*$(objm(t))$ does not satisfy $pair(t)$; otherwise it is *enabled*. Disabled GACLEntries do not contribute to the semantics and can be eliminated. We can simplify a list of GACLEntries by removing from it all disabled entries and simplify the EntryHead of enabled ones. This is precisely what the function *simplify*() does. It takes a GList l and returns a new GList containing only enabled entries whose EntryHead has been simplified. The parameter *obj* is needed to identify those GACLEntries that refer to other gacl's. Note that after simplifying GList l of gacl *obj*, all STriple's in the EntryHead of a GACLEntry in l should contain *obj* as the object modifier.

Given a consistent set A of authorizations, the *inverse* of A, denoted by A^I, is defined as the set of authorizations

$$\{\langle [S], [op] \rangle \mid \langle [S], [\overline{op}] \rangle \in A\}$$

and the *complement* of A, denoted by A^C, is defined as the set of authorizations

$$\left\{ \langle [S], [op] \rangle \;\middle|\; \begin{array}{l} S \text{ is the largest subset of } \textit{CInd} \text{ such that} \\ \text{if } \langle [S'], [\overline{op}] \rangle \in A \text{ then } S \cap S' = \emptyset \end{array} \right\}$$

We note that, unlike ordinary set complements, $(A^C)^C = A$ does not hold.

We observe also a subtle aspect of our semantics. That is, the authorization of a compound individual is only loosely related to the authorizations of its component individuals. For example, if both Alice and Bob are given read permission to a file, our semantics does not stipulate that the compound individual Alice $\wedge$ Bob be given the same permission. Although some may argue this is counter-intuitive, we believe it is best not to "built-in" too much into the semantics. The semantics should include only stipulations that are universal across all applications. Indeed, the above example could be easily handled by including an extra GACLEntry as follows:

$$\langle [\_x], [\mathsf{read}] \rangle \wedge \langle [\_y], [\mathsf{read}] \rangle \Rightarrow \langle [\_x \wedge \_y], [\mathsf{read}] \rangle.$$

5.2.1 Ordered GACL

A procedure for computing the semantics of an ordered gacl is shown in Figure 7. In the following, we briefly explain the steps involved.

First, the gacl is simplified by calling *simplify*(), which removes all disabled entries and simplifies the EntryHead of each enabled entry. We note that circularity can result if gacl's

```
function OrderedGACL (obj : ObjID) : set of Authorization {
  declare   A : set of Authorization
  A := ∅;
  obj.list := simplify(obj, obj.list)
  loop each e ∈ obj.list do
    if ∃ h : SubjOpPair ∈ head(e) such that A does not satisfy h
      next;
    case body(e) of
      STriple: let body(e) = obj :: p;
               A := A ∪ (expand(p) ∩ A^C);
      DTriple: let body(e) = default :: p;
               A := A ∪ (expand(p) ∩ A^C);
      ITriple: let body(e) = inherit o :: p;
               A := A ∪ (expand(p) ∩ semantics(o) ∩ A^C);
    end;
  end;
  return A;
}
```

Figure 7: Ordered gacl

```
function UnorderedGACL (obj : ObjID) : set of Authorization {
  declare   A1, A2 : set of Authorization
            SI, D, I : GList
  obj.list := simplify(obj, obj.list);
  SI := {e ∈ obj.list | body(e) ∈ STriple or body(e) is an always inherit}
  D  := {e ∈ obj.list | body(e) ∈ DTriple}
  I  := {e ∈ obj.list | body(e) is a demand inherit}
  A1 := smallest set of Authorization closed under SI;
  if A1 is inconsistent
    return error;
  A2 := smallest set of Authorization containing A1 and
        closed under SI ∪ D ∪ I;
  if A2 is undefined or inconsistent
    return error;
  else
    return A2;
  end;
}
```

Figure 8: Unordered gacl

mutually "depend" on one another. As an example, consider the following pairs of gacl's for object o_1 and o_2:

o_1	**declare**	ordered
	list	$\mathsf{o}_2 :: \langle[\mathsf{b}], [\mathsf{W}]\rangle \Rightarrow \langle[\mathsf{b}], [\mathsf{W}]\rangle$
o_2	**declare**	ordered
	list	$\mathsf{o}_1 :: \langle[\mathsf{a}], [\mathsf{R}]\rangle \Rightarrow \langle[\mathsf{a}], [\mathsf{R}]\rangle$

In constructing the semantics of gacl o_1, the semantics of gacl o_2 is needed for determining whether the only GACLEntry in o_1 is enabled. Therefore, the semantics of gacl o_2 must be available before that of gacl o_1. However, by a similar argument, the semantics of gacl o_1 is needed to determine the semantics of gacl o_2. This circularity results in an infinite recursion in our procedural semantics. There are two solutions to this problem: (1) The semantics can be modified to use an iterative incremental construction that terminates when no increment is observed. In other words, the procedure allows the semantics of o_1 and o_2 to be computed together in an iterative manner. (2) A syntactic restriction that forbids such circularity. We opted for solution (2) as it is less error-prone and much easier to comprehend for an authorization administrator.

We say that a gacl o *depends* on another gacl o' if there exist a GACLEntry in o whose EntryHead contains a STriple that refers to o'. An authorization specification is *well-formed* if no gacl o transitively depends on itself. Our semantics is only well-defined for well-formed authorization specifications.

A similar circularity problem can also surface in handling inheritance. If gacl o inherits from gacl o' and vice versa, an infinite recursion would result via the function *expand*(). We can disallow this by also defining that o depends on o' if o inherits from o'. Thus, a well-formed authorization specification does not have recursive dependence in its closure and inheritance properties.

After simplification, each GACLEntry in *obj*.**list** is examined one by one, proceeding from the first to the last in the list (cf. **loop each** construct).

Each GACLEntry belongs to one of three types, STriple, DTriple and ITriple, depending on its EntryBody. The handling of STriple and DTriple is essentially the same.[12] In each case, the new authorizations represented by an entry are given by *expand*(p) (where p is the SubjOpPair part of the EntryBody), and are merged into A after intersecting with A^C. The intersection guarantees that no conflicting authorizations from those already in A would be added. In other words, authorizations already in A take precedence over new authorizations. That is, conflict resolution is based on ordering.

For the case of Itriple, we only want to merge in authorizations represented by p that also belongs to the semantics of $objm(t)$. This is achieved by first "filtering" what is returned by *expand*(p) through the semantics of $objm(t)$ using an intersection before merging them into A.

5.2.2 Unordered GACL

A procedure for computing the semantics of an unordered gacl is shown in Figure 8. We explain the steps here.

First, as in the case of ordered gacl, we simplify the list part by calling *simplify*(). Then we separate out the GACLEntries into three different groups. The first group SI contains entries that should always be considered. The second group D contains all of the default entries, i.e., entries that should be considered only on demand. Similarly, the third group I contains inheritance entries that are only considered on demand.

As we mentioned earlier, the entries of an unordered gacl should be examined "together" to determine authorization. This is formalized by the notion of a *closed* set of authorizations we define below.

Let A be a set of authorizations and e a GACLEntry. We say that A is *closed under* e if the following holds: If for all p : SubjOpPair $\in head(e)$, A satisfies p, then

Case 1. $body(e) \in$ STriple

A satisfies $body(e)$;

[12] This confirms our earlier statement that in an ordered list, **default** entries are mainly for clarification purposes.

Case 2. $body(e)$ = **always inherit** $o :: p$

$expand(p) \cap semantics(o) \subseteq A$;

Case 3. $body(e)$ = **default** :: p

$expand(p) - A^I \subseteq A$;

Case 4. $body(e)$ = **demand inherit** $o :: p$

$(expand(p) \cap semantics(o)) - A^I \subseteq A$.

A is *closed under* a set E of GACLEntries if A is closed under each element of E.

Referring back to Figure 8, we note that A_1 must exist. This is because if two set of authorizations A and A' satisfy p, then $A \cap A'$ also satisfies p. From this, we can deduce that if both A and A' are closed under a GACLEntry $e \in SI$, then $A \cap A'$ is also closed under e.[13] Thus A_1 can be obtained by intersecting the collection of all sets of authorization closed under SI.

A_2, however, may not exist; and even if it does, may be inconsistent. As an example, consider the following gacl:

o	**declare**	
	list	**default**::$\langle$[a], [R]$\rangle$, **default**::$\langle$[a], [−R]$\rangle$

Both SI and I are empty, while D = {**default** :: $\langle$[a], [R]$\rangle$, **default** :: $\langle$[a], [−R]$\rangle$}. Since SI is empty, $A_1 = \emptyset$. Now consider A_2. It can easily be checked that both $\{\langle[\{a\}], [R^+]\rangle\}$ and $\{\langle[\{a\}], [R^-]\rangle\}$ are closed under D, and neither is smaller than the other. Thus A_2 does not exist. This is to be expected because both defaults are activated on demand and they specify contradictory authorizations.

5.3 Evaluation

A client request can be abstractly represented as a triple (s, op, o) where s is the identity of the client, *op* the access requested and o the object on which access *op* is desired. Given a request $req = (s, op, o)$ and a gacl for o such that *semantics*(o) $\neq$ failure, req is *granted* if there exists $\langle[S], [op]\rangle \in$ *semantics*(o) such that $s \in S$; req is *denied* if there exists $\langle[S], [\overline{op}]\rangle \in$ *semantics*(o) such that $s \in S$; and failure is returned otherwise.

Thus, a naive way of evaluating authorization is to first compute the semantics of a gacl and then apply the above. This, however, is highly inefficient unless the semantics is relatively static and a pre-compilation has been done. In general, only a small part of a gacl should be examined (e.g., the entries mentioning *op* for a request (s, op, o)) and a Prolog-type pattern matching algorithm can be used.[14]

[13] Technically, we need to show this for an arbitrary collection of sets, instead of just two sets.

[14] We note here that since a gacl may mention other gacl in its entries (e.g., in EntryHead or in EntryBody in a inheritance property), the evaluation process may need to examine parts of more than one gacl. This is similar to Prolog as well, in that the firing of a rule r may induce the firing of other rules corresponding to the subgoals of r.

P.exe	**declare**	ordered
	list	$\langle$[Alice, Bob], [−execute]$\rangle$,
		$\langle$[Dept], [execute]$\rangle$,
		$highload \Rightarrow \langle$[∗], [−execute]$\rangle$,
		inherit P.src::$\langle$[∗], [write]$\rangle$
P.doc	**declare**	anonymous
	list	P.exe::$\langle$[_x], [execute]$\rangle \Rightarrow \langle$[_x], [read]$\rangle$,
		$\langle$[Dept], [−write]$\rangle$,
		always inherit Doc::$\langle$[∗], [−read]$\rangle$,
		demand inherit Doc::$\langle$[∗], [write]$\rangle$
P.src	**declare**	
	list	$\langle$[Research], [read, write]$\rangle$,
		$\langle$[−Dept], [−write]$\rangle$
Doc	**declare**	ordered
	list	$\langle$[Dept], [read]$\rangle$,
		$\langle$[DocSys $\wedge$ Research], [∗]$\rangle$,
		default::$\langle$[∗], [−∗]$\rangle$

Figure 9: An Example Specification using GACL

Another way to speed up evaluation is to reduce group membership checks. As an example, consider the following gacl:

o	**declare**	ordered
	list	$\langle$[G_1], [−R]$\rangle$, $\langle$[G_2], [R]$\rangle$, $\langle$[G_3], [R]$\rangle$

and a request (a, R, o). Since this is an ordered gacl, evaluation proceeds sequentially from the first entry to the last. It is easy to see that a grant can be authorized if a can provide in order a nonmembership certificate for G_1, and then a membership certificate for either G_2 or G_3. Otherwise, a denial or a failure should be returned. Thus at least two membership checks are needed, which, in the worst case, can require two rounds of message exchanges. However, if information on group relationships are available, some savings are possible. For example, if it is known that $G_2 \subseteq G_1$, then the evaluation can skip over the G_2 entry all together and proceed directly to the G_3 entry. Such group relationship information can be provided as auxiliary information in an authorization specification or separately by the group servers.

5.4 An Example

Consider a set of objects {P.exe, P.doc, P.src, Doc}. P.exe, P.doc and P.src together constitute a software package with P.exe being the executable, P.doc the documentation, and P.src the source. Doc is a centralized documentation control system in which P.doc is a part. Alice and Bob are individual users while Research and Dept are groups. DocSys is a server responsible for maintaining the documentation control system (e.g., performing version control). Though DocSys is not an actual user, it is considered a user in our design. We consider only three types of access, namely, read, write and execute. *highload* is a system predicate whose (boolean) value is continuously updated by some system component that monitors the load of the system. For brevity, in the following, we refer to an entry by its position in the list. For example, with respect to gacl P.src, entry 2 refers to the entry $\langle[-\mathsf{Dept}], [-\mathsf{write}]\rangle$.

Consider gacl P.exe in Figure 9. Entries 1 and 2 are similar to those in ordinary ACL. Entry 1 specifies that both Alice and Bob are not permitted to execute P.exe, while entry 2 specifies that members of group Dept are allowed to execute P.exe. Entry 3 specifies that if the value of $highload$ is true, then no subject is allowed to execute P.exe. ($*$ stands for all subjects.) Entry 4 specifies that any subject who can write P.src can inherit the same access (i.e., write) to P.exe. Since "ordered" is declared, these entries should be examined in order from entries 1 to 4 in determining authorization. For example, Alice will be denied execute right for P.exe even if she belongs to Dept or has write access to P.src.

Consider gacl P.doc in Figure 9. Entry 1 specifies that any subject who has execute right for P.exe can also read P.doc. (_x is a variable that can be instantiated to any subject.) Entry 2 specifies that members of Dept cannot write P.doc. Entry 3 specifies that any subject who is denied read access to Doc will inherit the same denial to P.doc. Entry 4 specifies that any subject who has write access to Doc can inherit on demand the same access to P.doc. A demand inheritance is activated only if no other write authorization has been specified in other entries. For example, members of Dept would not be able to inherit their write access to Doc (even if they do have it) because of entry 2. Note that gacl P.doc is unordered, thus its entries must be considered together in making a determination. For example, if Alice has execute right to P.exe (cf. entry 1) but is denied read access to Doc (cf. entry 3), then a read request from Alice for P.doc would generate a failure as entries 1 and 3 together specify contradictory read authorizations for Alice.

The "anonymous" declaration does not affect the semantics of authorization. It indicates that an end server is willing to accept authorizations certified by an authorization server even without precise knowledge of the client making the request. For example, if a client other than Alice or Bob presents itself only as a member of Dept without precisely identifying who it is, it will still be acceptable to the end server and be granted read access.

Consider gacl P.src in Figure 9. Entry 1 specifies that members of Research can read and write P.src. Entry 2 specifies that any subject not belonging to Dept is denied write access to P.src. Again, gacl P.src is unordered. Thus a write request for P.src from any member of Research who is outside of Dept would generate a failure.

Consider gacl Doc in Figure 9. Entry 1 specifies that all members of Dept have read access to Doc. Entry 2 illustrates authorizations for compound subjects. A *compound subject* can informally be understood as a subject who has authority to act as each of its component subjects. Thus, entry 2 specifies that any subject who has authority to act both as DocSys and as a member of Research can be granted all accesses to Doc. Typically, a compound subject is constructed by *delegation*. For example, a member of Research who has obtained delegation from DocSys to act on behalf of DocSys is an instance of the compound subject DocSys $\wedge$ Research. Entry 3 specifies that by default, every subject should be denied all accesses. Since "ordered" is declared, this default serves as a negative catch-all, and provides the "denial by default" semantics of ordinary ACL. Defaults are typically used in an unordered gacl; its activation is then similar to that of demand inheritance. For an ordered gacl, the keyword **default** is optional; it serves as a comment. For example, the semantics of gacl Doc is unchanged if the **default** modifier is dropped from entry 3.

6 Discussion and Related Work

There are two distinct ideas underlying our design: (1) The use of GACL as a common representation of authorization requirements. (2) The use of authenticated delegation to effect authorization offloading from an end server to an authorization server.

The major strength of GACL is its expressiveness and the availability of a precise semantics. Its expressiveness is particularly useful in specifying authorization requirements for services that manage a large number of objects with complex dependencies. Its formal semantics facilitates the implementation of different evaluation strategies that can interoperate. The use of a declaration section is novel. It provides directives for choosing the most efficient evaluation strategy. For example, an unordered gacl can potentially make use of direct hashing in its evaluation, while an ordered gacl allows a partial evaluation strategy.

The language of policy base proposed in [15] is more general than GACL (in particular, it subsumes first-order logic) and has a much more abstract semantics. The GACL language is intended to be practical, and can indeed express the basic structural properties identified in [15], though not in their full generality. Moreover, the semantics of GACL is more procedural, as opposed to the declarative nature of the semantics of the language of policy base. The use of a declaration section also adds to the practicality of GACL.

Authenticated delegation has been used and studied in other works [1, 2, 5, 7, 10]. Most of these, with the notable exception of Neuman's [10], concentrate on the authentication and operational aspects of delegation rather than its application. The work reported in [1, 5] presents a formal understanding of authenticated delegation. In particular, it introduces a *handoff rule* that can be used to formally explain protocols for authenticated delegation. Gasser and McDermott [2] discuss how to carry out delegation in various contexts (e.g., user-host, process-process). The work by Neuman [10] is most relevant to ours. He describes a proxy-based method for performing authorization and accounting. A *proxy* is essentially an authenticated delegation. He describes several applications of proxies (e.g., capabilities, group servers) that are applicable in our design as well. One difference between our design and Neuman's method is that in his method, an authorization server is not *authoritative*, in the sense that an authorization server does not directly assert whether a subject can be granted access or not. Instead, it allows a client to act (in a restricted manner) as itself in requesting access at an end server (by granting the client a *restricted proxy*). The final authorization is carried out by the end server using acl's that contain entries specifying the authority of the authorization server. Our design can be easily adapted so that an authorization server pre-screens clients only, leaving the final authorization determination to an end server. Also, Neuman's focus is more on applications of proxies; the representation and evaluation issues involved in constructing a complete authorization service were not discussed.

On the more practical side, [12] studies a flexible authorization protocol for delegation, and discusses how it supports existing authorization models such as OSF DCE and SESAME. This protocol is based on a concept of proxy delegation, which as we pointed out above, is in the same spirit as our authenticated delegation. In [3], a distributed authorization model for WWW was proposed. The problem of "coordinated authorization" as studied there is a special case of authorization closure.

7 Conclusion

As WWW and Internet-based commerce become more and more popular, an authorization service, such as the one described in this paper, will be highly desirable. The two central problems of authorization are how it should be specified, and what protocols should be used to support it.

The specification problem is particularly acute for large services, e.g., for WWW sites serving large number of subscribers and documents with related authorization requirements. Existing approaches based on file system structure or simple acls do not scale well as the number of objects increases. The protocol problem is critical as more and more services are accessible via the Internet, which is far from secure, and open to attacks.

An authorization service can be used to relieve an end server of its routine authorization functions. Together with an authentication service, it facilitates the implementation of secure end services in a distributed system. Specifically, it enhances the overall security of a system by providing a well-defined, security-tested, basic building block to a service implementor.

A prototype implementation of a GACL interpreter has been finished, and is reported in [13]. It has an X-window user interface, and allows evaluation of GACL specifications. The current prototype has traded off the full generality of GACL (e.g., only acyclic specifications are allowed) for more efficient evaluation.

References

[1] M. Abadi, M. Burrows, B.W. Lampson, and G. Plotkin. A calculus for access control in distributed systems. *ACM Transactions on Programming Languages and Systems*, 15(4):706–734, September 1993.

[2] M. Gasser and E. McDermott. An architecture for practical delegation in a distributed system. In RSP90 [11], pages 20–30.

[3] J. Kahan. A distributed authorization model for WWW. In *Proceedings of INET'95*, 1995.

[4] J.T. Kohl, B.C. Neuman, and T.Y. Ts'o. The evolution of the Kerberos authentication system. In F. Brazier and D. Johansen, editors, *Distributed Open Systems*, pages 78–94. IEEE Computer Society Press, 1994.

[5] B. Lampson, M. Abadi, M. Burrows, and T. Wobber. Authentication in distributed systems: Theory and practice. *ACM Transactions on Computer Systems*, 10(4):265–310, November 1992.

[6] H.M. Levy. *Capability-Based Computer Systems*. Digital Press, 1984.

[7] J. Linn. Practical authentication for distributed computing. In RSP90 [11], pages 31–40.

[8] J. Linn. *Generic Security Service Application Program Interface, Version 2*. RFC 2078, January 1997.

[9] R. Molva, G. Tsudik, E. Van Herreweghen, and S. Zatti. *KryptoKnight* authentication and key distribution system. In *Proceedings of 2nd European Symposium on Research in Computer Security*, pages 155–174, Toulouse, France, November 23–25 1992. Springer Verlag.

[10] B.C. Neuman. Proxy-based authorization and accounting for distributed systems. In *Proceedings of 13th International Conference on Distributed Computing Systems*, pages 283–291, Pittsburgh, Pennsylvania, May 1993.

[11] *IEEE Symposium on Research in Security and Privacy*, Oakland, California, May 7–9 1990.

[12] J.T. Trostle and B.C. Neuman. A flexible distributed authorization protocol. In *Proceedings of Symposium on Network and Distributed Systems Security*, 1996.

[13] C.K. Wong and S.S. Lam. GACL language manual (44 pages). Technical report, Department of Computer Sciences, The University of Texas at Austin, August 1996.

[14] T.Y.C. Woo, R. Bindignavle, S. Su, and S.S. Lam. SNP: An interface for secure network programming. In *Proceedings of USENIX Summer Technical Conference*, Boston, Massachusetts, June 6–10 1994. Available from http://www.cs.utexas.edu/users/lam/NRL/.

[15] T.Y.C. Woo and S.S. Lam. Authorization in distributed systems: A new approach. *Journal of Computer Security*, 2(2,3):107–136, 1993.

[16] T.Y.C. Woo and S.S. Lam. A framework for distributed authorization (extended abstract). In *Proceedings of 1st ACM Conference on Computer and Communications Security*, pages 112–118, Fairfax, Virginia, November 3–5 1993.

[17] T.Y.C. Woo and S.S. Lam. Authentication for distributed systems. In Dorothy Denning and Peter Denning, editors, *Internet Besieged: Countering Cyberspace Scofflaws*. ACM Press and Addison-Wesley, 1997.