Designing an Authorization Service*

Thomas Y.C. Woo Simon S. Lam
Department of Computer Sciences
The University of Texas at Austin
Austin, Texas 78712-1188

Abstract

We present the design of an 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.

*Research sponsored by NSA INFOSEC University Research Program under contract no. MDA 904-94-C-6106. Technical report TR-93-29, Department of Computer Sciences, University of Texas at Austin, October 1994 (revised).

1 Introduction

Advances in networking 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. 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 following problems:

- *Authentication* and *authorization* — Before services can be rendered, a server must be ascertained of a client's identity and determine that the client's request can be honored according to some specified authorization policy. Similarly, a client needs to affirm the legitimacy of a server before proceeding with the service.

- *Accounting* — Consumption of service incurs a cost, monetary or otherwise. The cost is payable by a client as soon as the service has been delivered or a service agreement has been reached.

- *Auditing* — On-line scrutiny of all interactions between clients and servers may not be desirable or feasible. Thus selective client-server interactions may be logged for possible subsequent off-line examination. For example, if security violation is detected, a log can be used to reconstruct the sequence of interactions that led up to the violation.

Among these problems, authentication is the most basic, as well as the most studied. Much work has recently been done on authentication [3, 6, 9]. Its issues are fairly well-understood. In fact, several authentication services are already available, e.g., Kerberos from MIT [5, 17] (which has also been integrated as part of the OSF DCE Security Service [16]), SPX [18] from DEC, and KryptoKnight [12] from IBM.

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 [6, 9, 15]. However, most of these works, with the notable exception of [15], concentrate on the authentication aspect. Our study of authenticated delegation is for authorization purposes, and is similar to the notion of *proxy* in [15].

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. In Section 4, high-level specifications of the protocols are presented. In particular, we describe the use of authenticated delegation in protocol design. In Section 5, we first give an informal introduction to the language GACL. Then we present the formal syntax and semantics of the

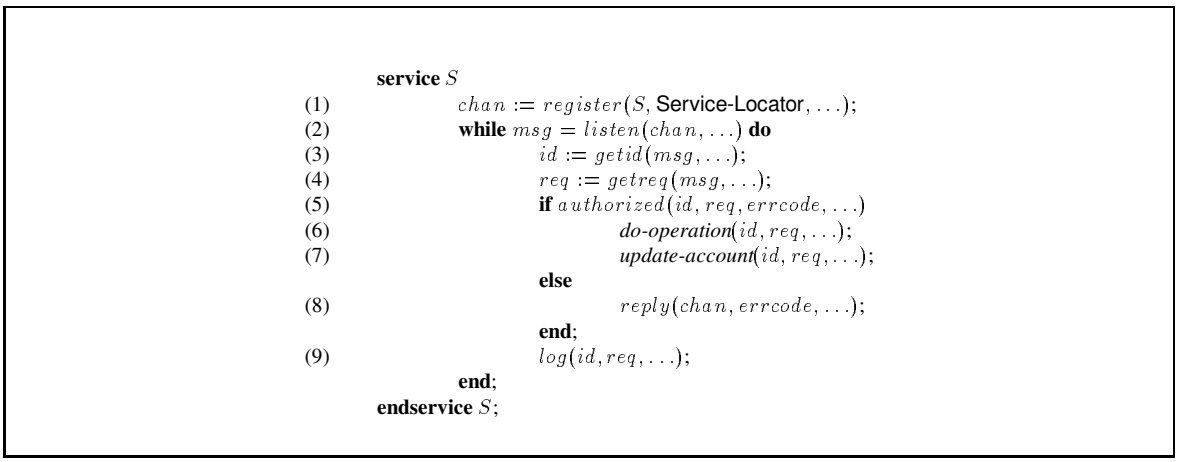

Figure 1: Structure of a typical service

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 for an Authorization Service

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*.[1]

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).[2]

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).

[1] This terminology is adapted from [15], 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.

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

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[3] 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., [5, 17]). There is even a proposal to standardize an *application program interface* for authentication services [10].

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 problem — 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 problem — Secure protocols are needed for offloading authorization from end servers to authorization servers, and for interactions among clients, authorization servers and end servers. These protocols are intended to make transparent the decoupling of authorization from end services.

[3] Some examples are GIDs (global unique ids) [7] and domain names [11].

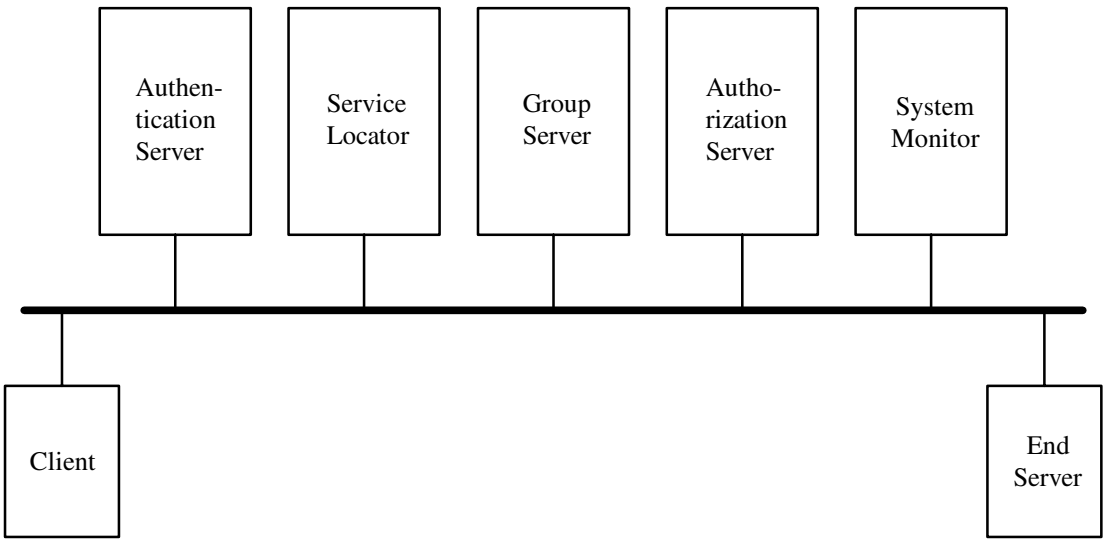

Figure 2: Architecture of authorization service

In Sections 4 and 5, 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 Subsection 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[4] 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).

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

- 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 *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.

We note that the above servers are only logically disjoint, they could easily be implemented as an integrated server or located on the same machine. To enhance efficiency, these servers can also be distributed and/or replicated. [5]

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 (cf. Kerberos [5, 17]).

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.

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.[6]

[5] Distribution refers to the partitioning of a distributed system into subsystems and the assignment of distinct servers to handle the subsystems [14]. Replication would bring in a number of standard distributed system problems (e.g., consistency) that need to be separately addressed.

[6] 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.

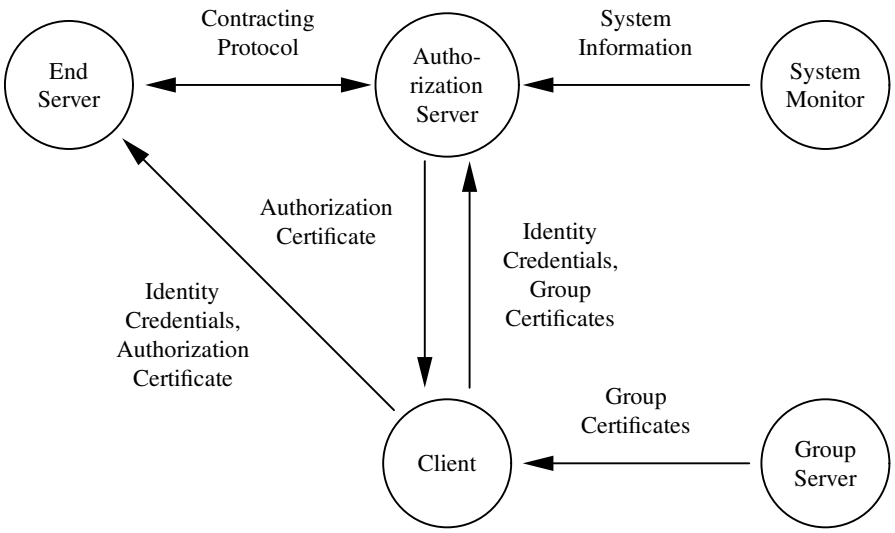

Figure 3: Message exchanges in our design

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 [15]. 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.[7] 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* [8, 13], and are beyond the scope of this paper.

[7] 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.[8] 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 implemented in [20], 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)	$S \rightarrow E$	:	$\{n_E, k_C, C\}_{k_S^{-1}}, \{\{n_C, n_E, C, E, k\}_{k_S^{-1}}\}_{k_E}$
(7)	$E \rightarrow C$	:	$\{\{n_C, n_E, C, E, k\}_{k_S^{-1}}\}_{k_C}$
(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., [12, 17, 18]).

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.[9] 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. 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

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

[8] 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).

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

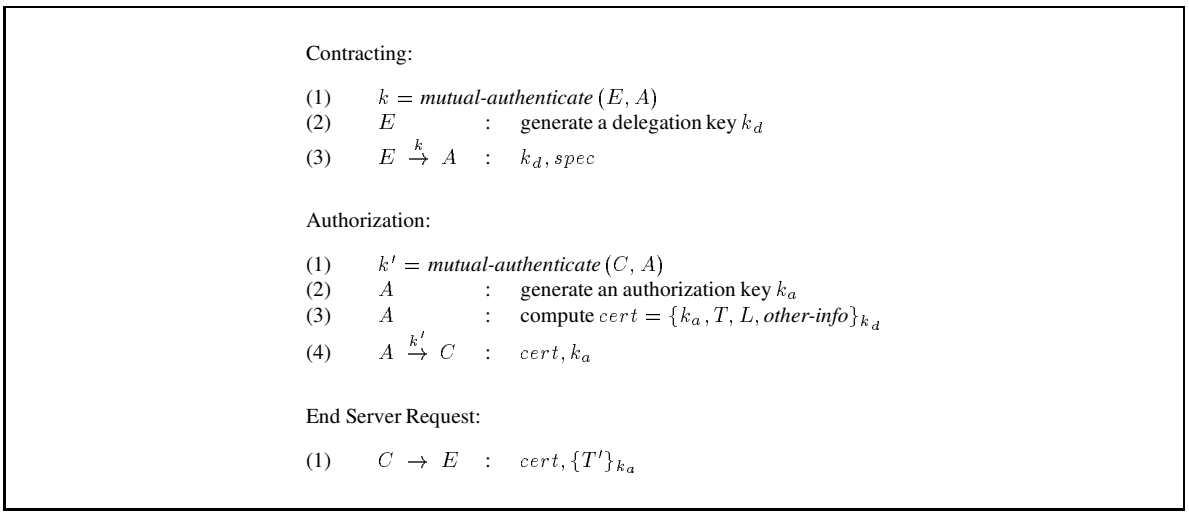

Figure 4: Use of authenticated delegation

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 = *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. □

5.1 Informal Introduction

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; each pair is made up of a subject identifier and 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 $r \in R$. Denial is implicit, i.e., it is implied by the absence of positive authorization in the list. As an example, consider the following acl for a file f: (Alice and Bob are individuals while Dept is a group)

$$\mathsf{f} : (\mathsf{Alice}, \{\mathsf{read}, \mathsf{write}\}), (\mathsf{Bob}, \{\mathsf{read}\}), (\mathsf{Dept}, \{\mathsf{write}\})$$

This acl specifies that Alice can be granted read and write accesses to f, and denied any other access to f. Similarly, Bob only has read access while all members in group Dept only have write access.

The key advantage of ACL is its straightforward semantics which is easy to understand. 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 [19], 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 to the "denial by default" style of authorization because a failure may be the result of an error in a specification. On the other hand, the language allows an authorization administrator to explicitly specify a catch-all "denial by default" if so desired.
- 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 *policy base* language introduced in [19]. We defer a comparison of the two to Section 6. In what follows, we give an informal introduction to GACL by example. The syntax of GACL is presented in Subsection 5.2, and its semantics in Subsection 5.3.

Consider Figure 5, where an example is specified using GACL. Each gacl is labeled by an object name and consists of two parts: (1) a *declaration* part identified by the keyword **declare**; and (2) a *list* part identified by the keyword **list**.

The declaration part contains a (possibly empty) list of predefined keywords that provide information for interpreting the gacl. In what follows, we discuss in detail only two such keywords: "ordered" and "anonymous". (Other keywords are mentioned in Subsection 5.5 on extensions of GACL.) An "ordered" declaration specifies that the list part of the gacl is an ordered list. That is, in determining authorization, its entries should be examined in a sequential order starting from the first to the last. By default, a gacl is interpreted as "unordered". For an unordered gacl, all entries in its list part should be examined "together" in making an authorization determination. This would be made clearer as we examine the example more closely below.

The list part contains a list of entries, some of which resemble those of ordinary acl's, while others are new. We informally explain their meanings below using the example in Figure 5.

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. In

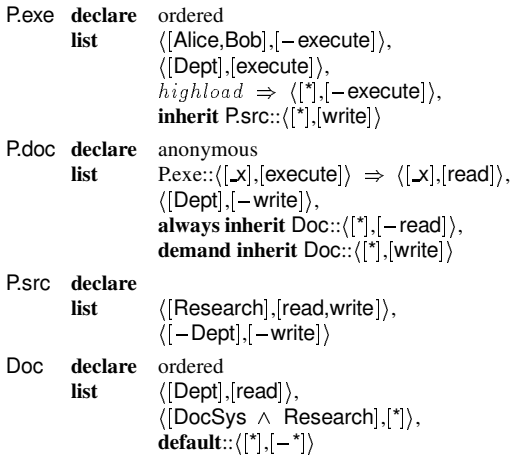

Figure 5: Specification of an example using GACL

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 ⟨[−Dept],[−write]⟩.

Consider gacl P.exe in Figure 5. 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 5. 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. That is, 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 an error 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 saying who it is, it will still be acceptable to the end server and be granted read access.

Consider gacl P.src in Figure 5. 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 an error.

Consider gacl Doc in Figure 5. 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

```
<SimplSubj>     :=  ["−"]<SubjID> | ["−"]<GroupID>
<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 6: A grammar for GACL

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.

5.2 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 6. 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 $-$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:

- 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 current gacl.
- All object modifiers occurring 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 also allow a shorthand notation using variables. A GACLEntry containing variables is interpreted as standing for the set of GACLEntry's obtained by instantiating the variables with all possible terms of compatible type. For example, if SubjID=$\{\mathsf{Alice}, \mathsf{Dept}\}$, then the first GACLEntry of P.doc in Figure 5 would stand for the set of GACLEntry's $\{\mathsf{P.exe} :: \langle[\mathsf{Alice}], [\mathsf{X}]\rangle \Rightarrow \langle[\mathsf{Alice}], [\mathsf{R}]\rangle$, $\mathsf{P.exe} :: \langle[\mathsf{Dept}], [\mathsf{X}]\rangle \Rightarrow \langle[\mathsf{Dept}], [\mathsf{R}]\rangle$, $\mathsf{P.exe} :: \langle[\mathsf{Alice} \wedge \mathsf{Dept}], [\mathsf{X}]\rangle \Rightarrow \langle[\mathsf{Alice} \wedge \mathsf{Dept}], [\mathsf{R}]\rangle\}$.

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. Thus, 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 [19] via the use of its "⇒", **default** and **inherit** constructs respectively.

5.3 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. Similarly, we define a set $Op = \{r^+, r^- \mid r \in \text{OpID}\}$. Thus, 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. 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) A mapping $grpmember$: GroupID $\mapsto$ $PowerSet(Ind)$. This gives the membership of each group. (2) A mapping Φ : Pred $\mapsto \{\mathsf{true}, \mathsf{false}\}$ with the property that $\Phi(\mathsf{T}) = \mathsf{true}$.

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 a $\wedge$ b and the individual c are allowed to perform 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

```
function members (s : SubjExpr) : set of CInd {
    if s ∈ CmpdSubj
        let s = s_1 ∧ ... ∧ s_n;
        return {i_1 ∧ ... ∧ i_n | i_1 ∈ members(s_1), ..., i_n ∈ members(s_n)};
    else
        case s of
            Individual: return {s};
            GroupID:    return grpmember(s);
            *:          return CInd;
            −s':        return CInd − members(s');
        end;
    end;
}

function expand (p : SubjOpPair) : set of Authorization {
    declare  S : set of SubjOpPair; A : set of Authorization
    let p = ⟨[subjlist], [oplist]⟩;
    S := {⟨[s], [op]⟩ | s ∈ subjlist, op ∈ oplist};
    A := {⟨[∅], [op]⟩ | op ∈ OP};
    for each ⟨[s], [op]⟩ ∈ S do
        A := (A − {⟨[R], [op]⟩ | ⟨[R], [op]⟩ ∈ A})
             ∪ {⟨[members(s) ∪ R], [op]⟩ | ⟨[R], [op]⟩ ∈ A};
    end;
    return {⟨[X], [y]⟩ ∈ A | X ≠ ∅};
}

function simplify (obj : ObjID; l : GList) : GList {
    declare  ℓ : GList
    loop each e ∈ l do
        if e is disabled
            next;
        remove from head all p ≠ ⊤ ∈ Pred;
        remove from head all t ∈ STriple such that obj ≠ objm(t);
        ℓ := ℓ, head ⇒ body;
    end;
    return ℓ;
}
```

Figure 7: Some basic functions

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:

$$semantics\ (obj) = \begin{cases} OrderedGACL\ (obj) & \text{if ordered} \in obj.\textbf{declare} \\ UnorderedGACL\ (obj) & \text{otherwise} \end{cases}$$

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 7. 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 GACLEntry's do not contribute to the semantics and can be eliminated. We can simplify a list of GACLEntry 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 GACLEntry's 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

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

We note that, unlike ordinary set complements, $A \cup A^C$ may not be equal to $CInd$. And neither does $(A^C)^C = A$ hold.

Ordered gacl

A procedure for computing the semantics of an ordered gacl is shown in Figure 8. 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 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.

```
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 8: Ordered gacl

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.[10] 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.

Unordered gacl

A procedure for computing the semantics of an unordered gacl is shown in Figure 9. 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 GACLEntry's 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.

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

```
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 9: Unordered gacl

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 $body(e)$;

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 GACLEntry's if A is closed under each element of E.

Referring back to Figure 9, 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.[11] 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**::⟨[a],[R]⟩, **default**::⟨[a],[−R]⟩

Both SI and I are empty, while $D = \{$**default** $:: \langle[\mathsf{a}], [\mathsf{R}]\rangle,$ **default** $:: \langle[\mathsf{a}], [-\mathsf{R}]\rangle\}$. Since SI is empty, $A_1 = \emptyset$. Now consider A_2. It can easily be checked that both $\{\langle[\{\mathsf{a}\}], [\mathsf{R}^+]\rangle\}$ and $\{\langle[\{\mathsf{a}\}], [\mathsf{R}^-]\rangle\}$ are closed

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

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.

5.4 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 $semantics(o) \neq$ error, 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 fail 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.[12]

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.

5.5 Extensions

In this subsection, we briefly describe several extensions to the GACL language. Most of these relate to new types of declaration.

1. A special subject identifier OWNER can be introduced. This stands for the owner of an object. It can be interpreted as a mapping from ObjID to a SubjExpr that does not contain negation. With the use of OWNER, a new declaration, namely, "owner", can be introduced. This specifies that if a request comes from the owner of an object o, then evaluation of gacl o should start by first examining entries concerning OWNER first. In other words, if "owner" is declared, OWNER entries take precedence over other entries in a gacl.

2. In the semantics presented, there is no built-in method for resolving inconsistent authorizations for an unordered gacl. This can be easily remedied if conflict resolution information can be explicitly stated in the declaration section. This is achieved by using *preference* declarations. Two types of preference declaration can be used: operation preference and group preference. An example of an operation preference is $-R > R$, which specifies that a negative authorization on R should be preferred over a

[12] 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.

positive one. An example of a group preference is $G_1 > G_2$, which says that an element in $G_1 \cap G_2$ should obey an authorization from G_1 instead of a contradictory one (if one exists) from G_2.

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 [19] 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 [19], 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 [2, 4, 6, 15]. Most of these, with the notable exception of Neuman's [15], concentrate on the authentication and operational aspects of delegation rather than its application. The work reported in [2, 6] 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 [4] discuss how to carry out delegation in various contexts (e.g., user-host, process-process). The work by Neuman [15] 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.

7 Conclusion

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.

Authorization in distributed systems is still a relatively young area. Many issues remain to be explored and studied. The design proposed in this paper is a first attempt at identifying and solving some of the problems.

In this paper, we have omitted discussion of many of the more practical details due to space limitation. For example, the problems of consistency (due to cache invalidation and certificate expiration), group membership maintenance and propagation of authorization must be addressed in an implementation.

A prototype implementation of our design is currently under way. We have finished implementing an authentication substrate upon which our authorization service operates. We are concerned with finding efficient evaluation strategies for GACL. We plan to report our implementation results in a future paper, which would also address the practical details mentioned above.

References

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

[2] M. Abadi, M. Burrows, B.W. Lampson, and G. Plotkin. A calculus for access control in distributed systems. Technical Report 70, Systems Research Center, Digital Equipment Corporation, February 1991. An abbreviated version appeared in Advances in Cryptology — CRYPTO '91, pages 1–23, Santa Barbara, California, August 11–15 1991.

[3] M. Gasser, A. Goldstein, C. Kaufman, and B.W. Lampson. The Digital distributed system security architecture. In *Proceedings of the 12th National Computer Security Conference*, pages 305–319, Baltimore, Maryland, October 1989.

[4] M. Gasser and E. McDermott. An architecture for practical delegation in a distributed system. In *Proceedings of the 11th IEEE Symposium on Research in Security and Privacy* [1], pages 20–30.

[5] J.T. Kohl and B.C. Neuman. The Kerberos network authentication service: Version 5 draft protocol specification. April 1993.

[6] B. Lampson, M. Abadi, M. Burrows, and T. Wobber. Authentication in distributed systems: Theory and practice. In *Proceedings of the 13th ACM Symposium on Operating Systems Principles*, pages 165–182, Asilomar Conference Center, Pacific Grove, California, October 13–16 1991.

[7] B.W. Lampson. Designing a global name service. In *Proceedings of the 4th ACM Symposium on Principles of Distributed Computing*, August 1985.

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

[9] J. Linn. Practical authentication for distributed computing. In *Proceedings of the 11th IEEE Symposium on Research in Security and Privacy* [1], pages 31–40.

[10] J. Linn. Generic Security Service Application Program Interface: Internet draft. June 11 1991.

[11] P. Mockapetris. Domain names — concepts and facilities. RFC 1034, November 1987.

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

[13] S.J. Mullender and A.S. Tanenbaum. The design of a capability-based distributed operating system. *The Computer Journal*, 29(4):289–299, 1986.

[14] B.C. Neuman. Scale in distributed systems. In *Readings in Distributed Computing Systems*. IEEE Computer Society Press, 1992.

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

[16] W. Rosenberry, D. Kenny, and G. Fisher. *Understanding DCE*. O'Reilley & Associates, Inc., 1992.

[17] J.G. Steiner, C. Neuman, and J.I. Schiller. *Kerberos*: An authentication service for open network systems. In *Proceedings of the USENIX Winter Conference*, pages 191–202, Dallas, TX, February 1988.

[18] J.J. Tardo and K. Alagappan. SPX: Global authentication using public key certificates. In *Proceedings of the 12th IEEE Symposium on Research in Security and Privacy*, pages 232–244, Oakland, California, May 20–22 1991.

[19] T.Y.C. Woo and S.S. Lam. Authorization in distributed systems: A formal approach. In *Proceedings of the 13th IEEE Symposium on Research in Security and Privacy*, pages 33–50, Oakland, California, May 4–6 1992.

[20] T.Y.C. Woo, R. Bindignavle, S. Su, and S.S. Lam. SNP: An interface for secure network programming. In *Proceedings USENIX Summer Technical Conference*, Boston, June 1994.