Distributed access control and the prototype of the Mojoy trust policy language

Huang, Chenxi

Distributed Access Control and the Prototype of the Mojoy Trust Policy Language

Chenxi Huang

Department of Computer Science

University of Durham

Durham DH1 3LE

England

08 Nov 2004

Thesis submitted for the degree of MSc

Abstract

In a highly distributed computing environment, people frequently move from one place to another where the new system has no previous knowledge of them at all. Traditional access control mechanisms such as access matrix and RBAC depend heavily on central management. However, the identities and privileges of the users are stored and administered in different locations in distributed systems. How to establish trust between these strange entities remains a challenge. Many efforts have been made to solve this problem. In the previous work, the decentralised administration of trust is achieved through delegation which is a very rigid mechanism. The limitation of delegation is that the identities of the delegators and delegatees must be known in advance and the privileges must be definite. In this thesis, we present a new model for decentralised administration of trust: *trust empowerment*. In *trust empowerment*, trust is defined as a set of properties. Properties can be owned and/or controlled. Owners of the properties can perform the privileges denoted by the properties. Controllers of the properties can grant the properties to other subjects but cannot gain the privileges of the properties. Each subject has its own policy to define *trust empowerment*. We design the Mojoy trust policy language that supports *trust empowerment*. We give the syntax, semantics and an XML implementation of the language. The Mojoy trust policy language is based on XACML, which is an OASIS standard. We develop a compliance checker for the language. The responsibility of the compliance checker is to examine the certificates and policy, and return a Boolean value to indicate whether the user's request is allowed. We apply our new model, the language and the compliance checker to a case study to show that they are capable of coping with the trust issues met in the distributed systems.

Acknowledgements

Many thanks to my supervisors Jie Xu and Keith Bennett. Many thanks to my friends who offered much help to me during my study in the University of Durham.

Table of Contents

List of Figures

List of Tables

Declaration

No material contained in this thesis has previously been submitted for a degree in this or any other university.

Statement of copyright

1 Traditional Access Control

The research of access control began from the early 1960s primarily centred in database management and operating systems. The objective was to protect system resources from unauthorised access.

1.1 Authentication, authorisation and access control

According to Apache definition [28], authentication is the procedure of verifying the users are who they claim they are. This is usually done by verifying the username/password, public key/private key, smart cards, or some other biological methods, such as voice recognition or fingerprints, to prove the identity.

Authorisation is to inspect whether the identified user has the permission to perform the specific action to the resource. This is done by checking the user's roles, privileges or attributes. Authorisation is analogous to checking tickets at the entrance of the cinema. Authentication and authorisation are closely related and in most implementations inextricable.

Access control is a much more general way of talking about controlling access to resources. Access decisions are made based on arbitrary conditions, such as network IP address, the time of day, the attributes of the user, or the version of the browser, etc. It is analogous to closing the door at a specific time or only letting people in by their ages.

1.2 MAC and DAC

The efforts of research and development on the part of the United States Department of Defense (DoD) over a period of twenty plus years formed a set of security criteria, criteria interpretations, and guidelines. It was the best known US computer security standard: the Trusted Computer System Evaluation Criteria (TCSEC). It contains security features and assurances, exclusively derived, engineered and rationalized based on DoD security policy, created to meet one major security objective – preventing the unauthorised observation of classified information.

The TCSEC has defined two types of access control: Discretionary Access Control (DAC) and Mandatory Access Control (MAC).

1.2.1 DAC

As defined in the TCSEC and commonly implemented, DAC [45] is "A means of restricting access to objects based on the identity of subject and/or groups to which they belong. The controls are discretionary in the sense that a subject with a certain access permission is capable of passing that permission (perhaps indirectly) on to any other subject (unless restrained by mandatory access control)" [8]. Since its appearance in 1983, DAC have been perceived as being technically correct for both commercial and civilian government security needs.

As the name implies, DAC permits the granting and revoking of access privileges to be left to the discretion of the individual users. A DAC mechanism allows users to grant or revoke access to any of the objects under their control without the intercession of the system administrator.

To illustrate the model of DAC, we need to first clarify several basic concepts. A subject is a process/program in the system; it can create other subjects; it is launched by a user. A subject has only one user as its owner, but a user can have multiple subjects. The relations between subjects, objects and access rights can be represented by an access matrix [46]. Subjects are represented in rows and objects are represented in columns. The cells contain the access rights assigned to the subjects that can be performed on the corresponding objects. The access rights are defined as $A_{s,o}$, where *s* is the subject and *o* is the object.

- If *action* $\in A_{s,o}$, then *s* can perform *action* on *o*.
- If the copy flag is set, i.e., **action* $\in A_{s,o}$, then *s* can add *action* to any cell in the column of *o*, i.e., assign access rights of *o* to any subject.
- If *owner* $\in A_{s,o}$, then *s* is the *owner* of *o*, *s* can assign access rights of *o* to any subject, i.e., add action to any cell in the column of *o*.
- If *owner* $\in A_{s,o}$, then s can revoke any subject's access rights to *o*, i.e., remove any "unprotected" action in the cells of the column of o.

For example, Table 1-1 embodies an access matrix for users and files (*r* stands for *read*, *w* for *write*, *a* for *append*, *d* for *delete*, *o* for *owner*, and *e* for *execute*). As we can see from the table, $r \in A_{Alice,File1}$, then Alice can read File1. $o \in A_{Alice,File1}$, then Alice can modify the actions in the cells of the column File1.

Table 1-1 access matrix

	File1	File2	File3	File4
Alice	rwo		rwd	
Bob	r	ro		
Clare				reo
Duncan	rwad			

An access matrix is usually sparse and can be implemented in several ways:

- Capabilities. A capability specifies what action a subject could perform on the designated object.
- Profiles. A profile contains a list of objects associated with a subject.
- Access Control List (ACL). ACL is the most desirable implementation of access matrix. An ACL contains a list of users/groups and their access rights to the designated objects.
- Protection Bits. It is the protection mechanism adopted by UNIX file system [44]. The creator of the file is the owner. The owner set the *protection bits* to indicate whether the owner, the group, or everyone could have the specific access right of the file.
- Passwords. The user gains a specific access right to the object by providing corresponding password.

However, in many organizations, the end users do not "own" the information for which they are allowed to access. For these organizations, the corporation or agency is the actual "owner" of the system objects as well as the programs that process it. This brings in MAC.

1.2.2 MAC

Mandatory Access Control was defined in the TCSEC as "A means of restricting access to objects based on the sensitivity (as represented by a label) of the information contained in the objects and the formal authorization (i.e. clearance) of subjects to access information of such sensitivity" [8].

MAC applies where protection decisions must not be decided by the object owner and must be enforced by the system. It is aimed to solve the problems that were unable to be dealt with by DAC.

The concept of MAC was first formalized as the Bell-LaPadula model by Bell and LaPadula [47]. Sandhu chose the essential assets from it and drew a minimal model, which was called BLP [49]. In BLP, MAC policies are expressed through the security labels attached to subjects and objects, which are called *security clearance* and *security classification* respectively. Security labels cannot be changed once they are assigned. Users do not have control over security labels. In order to gain access, users must first be authorised by the DAC access matrix, then pass the BLP MAC rules:

- Simple-Security Property: subject s can read object o only if $\lambda(s) \geq \lambda(o)$.
- *-Property: subject s can write object o only if $\lambda(s) \leq \lambda(o)$.

Note: λ denotes the security label. *Write* denotes "append" or "write only", and must not be interpreted as "read and write".

BLP MAC rules are defined as "only if" which means they are necessary but not sufficient conditions for access. Additional actions, such as create and delete, can be constrained by *-Property because they are similar to write.

Bell-LaPadula model was motivated by the purpose of confidentiality. It limits users to read downward (files below user's security level) and write upward (files above user's security level).

Another similar model, Biba model was motivated by the purpose of integrity [48]. The rules are:

- Simple-Integrity Property: subject s can read object o only if $\omega(s) \leq \omega(o)$.
- Integrity *-Property: subject s can write object o only if $\omega(s) \geq \omega(o)$.

Note: ω denotes the integrity label. Write denotes "append" or "write only".

The central notion is that low-integrity information is not permitted to flow to high-integrity objects, while the opposite is permitted. BLP model and Biba model can be combined together:

- Subject s can read object o only if $\lambda(s) \geq \lambda(o)$ and $\omega(s) \leq \omega(o)$.
- Subject s can write object o only if $\lambda(s) \leq \lambda(o)$ and $\omega(s) \geq \omega(o)$.

It is called composite model. It is very popular and has been implemented in several operating systems, databases and network products.

1.3 Role Based Access Control

The notion of Role Based Access Control (RBAC) emerged in the 1970s. It was first

formalized by D. Ferraiolo and R. Kuhn in 1992 [6]: Access permissions are assigned to roles and roles are assigned to users. As a bridge between permissions and users, role greatly simplifies corporate security management. Users are assigned different roles to reflect their positions and responsibilities. Roles are assigned different permissions to reflect organizational security policy changes. An investigation conducted by NIST (National Institute of Standards and Technology) on the security requirements of twenty eight different organizations showed that organizations desired access privileges be based on the position held by each person within the organization [19]. Organizations would like to maintain the access privileges in accordance with its security policies rather than at personal discretion. RBAC has the flexibility to meet these criteria.

UNIX/Linux users often find that the concept of UNIX/Linux group is similar to that of RBAC. The primary difference between groups and roles is that a group is only a collection of users. You need to go through the whole system to collect all the permissions that have been assigned to a group, for instance, in the UNIX file system, the administrator needs to traverse all the files in the file system to gather the permissions of a user/group, which will take a very long time. A role is both a collection of users on one side and a collection of permissions on the other side. Roles effectively and dynamically connect the two.

Sandhu et al. further classified RBAC into four sub categories, RBAC0, RBAC1, RBAC2 and RBAC3 [7]. RBAC0 is the basic model and the minimal requirement for any system that supports RBAC. RBAC1 and RBAC2 both embrace RBAC0 but extend with different features. RBAC3 combines RBAC1 and RBAC2 and, by transitivity, RBAC0 (Figure 1-1).

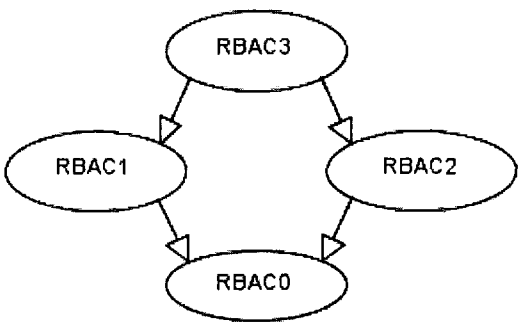

Figure 1-1 RBAC0 – RBAC3 relations

There are four elements in RBAC0; they are users (U), roles (R), permissions (P) and

sessions (S). Permissions and roles are many-to-many relation; users and roles are also many-to-many relation. A user could establish one or more sessions with the server. The user is the owner of the session and has the full control over it. He/she could have one or more roles active in one session at the same time and could dynamically change active roles. A session can only be associated with one user (the owner). Therefore a user could have different roles (permissions) active at the same time, or in one session at different times. This feature supports the principle of least privilege, which requires that a user should be given no more privilege than necessary to perform a job.

RBAC1 extends RBAC0 with the concept of hierarchies. Hierarchy is a very common requirement of the management of large organizations. It mirrors the inner security structure of large organizations. Figure 1-2 shows a typical diagram of role hierarchies. The role *member* is in the lowest level. *Database administrator* and *programmer* are in the middle and inherit all the privileges from *member*. *Supervisor* is the highest-level role and inherits all the privileges of *database administrator* and *programmer*. Sometimes we need to limit the scope of inheritance. For instance, *database administrator* and *programmer* want to keep some of the permissions private and prevent *supervisor* from inheriting them. This can be solved by adding new roles into the structure (Figure 1-3, *database administrator'* and *programmer'*). Under this situation, *database administrator'* and *programmer'* are often referred to as private roles.

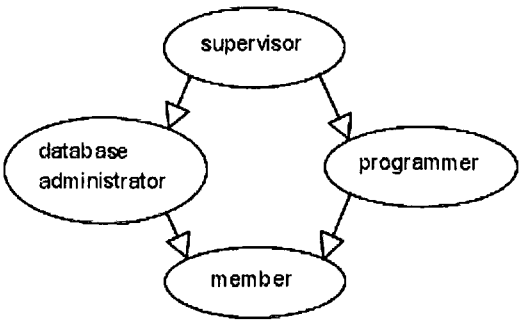

Figure 1-2 an example of role hierarchies

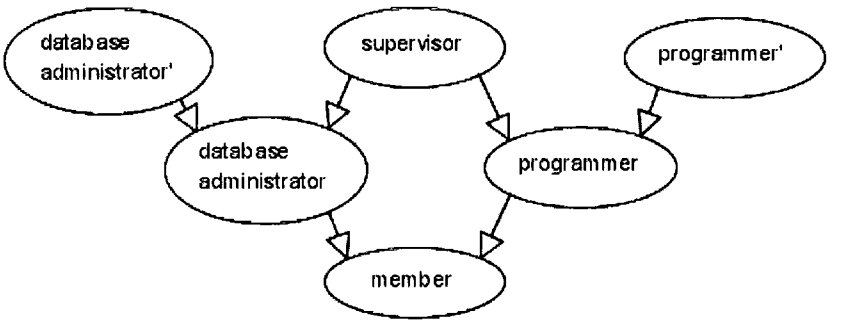

Figure 1-3 an example of private roles

RBAC2 extends RBAC0 with the concept of constraints. The first and most frequently mentioned constraint is mutually exclusive roles, i.e., the same user can be assigned at most one role at one time from a set of mutually exclusive roles. This feature supports the notion of separation of duties. For instance, *examiner* and *examinee* are two mutually exclusive roles and they cannot be assigned to one person at the same time. An extension of this model is that the mutually exclusive permissions cannot be assigned to the same role at the same time. The second constraint is cardinality constraint, i.e., one role can have only a specific maximum number of members and vice versa. For instance, there is only one person who has the role of manager in a department at one time. Correspondingly, the number of roles a permission can be assigned to can have cardinality constraint to control the distribution of powerful permissions. The third constraint is prerequisite roles, i.e., a user can be assigned role B if the user already has role A. This constraint is based on competency; a user should have a junior role in order to be assigned a senior role. However, if the same person has been assigned two or more identities, or the same operation could be accomplished by two different permissions, then separation and cardinality constraint become ineffective.

RBAC3 combines RBAC1 and RBAC2 together. This could introduce several new issues. Constraints can be applied to role hierarchies. For instance, in Figure 1-3 *database administrator'* and *programmer'* can be declared as mutually exclusive. They cannot be inherited by the same role or assigned to the same user. Real programmers should be assigned the role *programmer'*, not *programmer*. Therefore, the role *programmer* should have a maximum cardinality constraint of zero.

1.3.1 RBAC Extensions

Apart from the above basic RBAC models, there are several extensions of RBAC.

Parameterised RBAC appends parameters to roles. The privileges of roles can be refined during activation time by setting the parameter values. For example, instead of specifying every form of the privileges such as *read haematology*, *read biochemistry* and *read microbiology*, we can specify the privileges in the form of *read(name)*, where *name* can be replaced by *haematology*, *biochemistry* and *microbiology*. Parameterised RBAC also introduces problems for role hierarchies and constraint. Parent role parameters should not be significantly different from child roles and administrators need more information than the role name to place constraint.

Different from common RBAC models which allow subjects to do, obligation policy requires subjects must or must not do to the targets. Ponder is a RBAC policy language that supports obligation [14].

Positive permissions specify what a subject is allowed to do. On the contrary, negative permissions specify what a subject cannot do. Combining them together could be thorny because some actions can be both allowed and prohibited at the same time. This involves a priority order of the rules.

Centralised RBAC system is unable to meet the new requirements of distributed large scale organisations. For instance, it is difficult to know the permissions of a role assigned to a user from a different security domain; different portions of organisations need to maintain and modify security policies locally. Many efforts have tried to extend RBAC to support decentralised management [2][33].

1.3.2 Delegation

Delegation facilitates decentralised management of RBAC by allowing a user/role to authorise other users/roles with part/all of the privileges the user/role has. Delegator is the subject who gives out privileges and delegatee is the subject who receives privileges. For instance, Alice grants Bob the role *employee*. Alice is the delegator, Bob is the delegatee and *employee* is the delegated role. Delegation is closely related with revocation, i.e., the delegator can revoke a specific privilege from the delegatee.

Delegations are finely divided into several sub-categories [31].

- Permanence. The delegator permanently delegates all his roles to the delegatee. Delegator loses his roles and delegatee receives the full power of the delegator's roles. Delegator cannot take his roles back, except through the help of the administrator. On the contrary, temporary delegation allows the delegator to delegate his roles for a short period. After that time, delegation expires and delegator regains all his prior roles.
- Monotonicity. A monotonic delegation means delegator maintains all of his/her roles and permissions after delegation. A non-monotonic delegation means delegator loses all of his/her roles and permissions after delegation. Delegator can regain his/her roles and permissions by revoking the delegation.
- Totality. Total delegation delegates all the permissions of a role to the delegatee. Partial delegation only delegates a subset of the permissions of a role to the delegatee.
- Administration. In the self-acted delegation, the responsibility of the administration of delegation is on the delegator himself. In the agent-acted delegation, the administration is on a nominated third party.
- Levels of delegation. Delegation level can be specified by a number. Single step delegation prevents the delegated privileges to be further delegated. Multi-step delegation allows the delegatee to further delegate the privileges to other subjects.
- Multiple delegations. This type of delegation allows the delegator to delegate a role to multiple subjects at the same time.
- Agreements. Bilateral agreement is a contract, which contains the specification of the delegation, accepted by both the delegator and the delegatee. On the other hand, unilateral agreement is a one-way decision. The delegator decides to delegate the role and the delegatee has to accept it.
- Revocation. Revocation is divided into cascading revocation and grant-dependency revocation. Cascading revocation is the indirect revocation as a result of the revocation of other roles. A supporting role is the role that the delegatee owns prior to delegation. If the delegatee loses the supporting role, then he loses the delegated role. A sponsoring role is the role that the delegator owns in order to delegate. If the delegator loses the sponsoring role, then he loses the ability to delegate, the delegatee loses the delegated role and further delegations are also revoked. Grant-dependent delegation only permits the delegator to

revoke the delegation. Grant-independent delegation permits anyone who has the sponsoring role to revoke the delegation.

1.4 Summary

DAC allows the owner to manage his resource at his own will. MAC compels the resources to be managed by the system regardless whoever the owner is. RBAC supplements the drawbacks of both DAC and MAC. Users assigned roles and roles are assigned privileges. This change brings more flexibility. The basic concept of RBAC is extended with hierarchies, constraint, parameters and delegation. Traditional RBAC is centrally administered and satisfies the security requirements of locally managed organizations. However, in a distributed computing environment, where organizations need to frequently cooperate and coordinate with each other, a decentralised mechanism is desperately needed.

2 Distributed Access Control

2.1 New challenges and research issues

Most traditional access control systems depend on access matrix [46] and RBAC. Access matrix and RBAC are effective where identities and privileges are managed in one central point and the trust relationships are straightforward. However, in a distributed environment, where the users, the privileges and the objects are scattered and managed in different locations, access matrix and RBAC become inept to meet the increasingly complex security requirements [1]. For instance, organizations need to specify complicated situations such as time and location constraints, advanced RBAC models and dynamic trust relationships, etc in the policy. These cannot be clearly described by an access matrix. Some systems hard code the access control component into the application, this approach is inflexible because the only method to update the policies is to rewrite, recompile and redeploy the application, this is time consuming and costly. The major issues and requirements are observed and discussed in [4][10][9][32]:

- Large scale. The interconnected systems are composed of applications, resources and users from various locations that are geographically dispersed. Some of them are already known and trusted, some are unknown and therefore not trusted.
- Autonomous. As there is no central control point, each security domain should be able to independently specify, manage, and enforce its own security policies.
- Complex policies. Traditional access control mechanisms only consider simple conditions such as username/password. Nowadays we need to express more complex conditions such as time, location restriction, users' attributes, etc.
- Evolution. Because the nature of large scale, applications and systems change consistently. Security policies need to evolve accordingly to adapt to the changes.
- Cross security boundaries. Because of the nature of large scale, distributed systems often span several security domains. These domains maintain their own access control systems, potentially different RBAC systems. A mechanism must be developed to connect and coordinate these different RBACs.

- Decentralisation. The management of the access control system must be decentralised. Currently, the most popular mechanism is delegation which is not perfect. New mechanisms are required.

2.2 Decentralisation and delegation

Contrary to conventional centralised administration, the decentralised administration of trust is to administer users and their privileges in different locations. Until now, most of the existing solutions depend heavily on delegation [31]. The central notion of delegation is that a subject (delegator) can grant a subset of his privileges to another subject (delegatee), and the delegatee can further delegate the privileges to other subjects. Delegation facilitates decentralisation of administration by spreading privileges.

However, delegation has been criticised [10], the major disadvantages of delegation are:

- The identities of the subjects (both delegators and delegatees) must be known in advance.
- The delegator must delegate no more than what he/she has, and the privileges must be known in advance.

However, sometimes the identities of the delegators cannot be known. Also under certain situations, delegators do not own the privileges. For instance, according to The Medical Records Confidentiality Act [15], anonymised patients' data can be disclosed to researchers from a certified Health Information Service for research purposes only. Researchers have certificates from the Health Information Service to prove their identities. A Health Information Service is further certified by a national health authority. The identity of the Health Information Service cannot be known in advance and it does not have the privileges to access patients' record. Therefore delegation is unable to solve this kind of problems. A new approach is desperately demanded.

2.3 Thesis contribution

In this thesis, we present a new trust conveyance model, *trust empowerment*, to compensate for the disadvantages of delegation. The core principle of *trust empowerment* is that the identities of the delegators do not have to be known in advance but they are proved by their attributes. The delegators do not have to possess all the privileges in order to grant them. This new mechanism is embodied through our proposed trust policy language, which is partly based on the achievements

of XACML [13]. The credibility of the subjects is totally and finely controlled by the service provider. A corresponding compliance checker is also developed for the policy language.

The thesis is organized as follows. Chapter 2 discusses related work on distributed access control. Chapter 3 discusses the definition of the trust policy language. Chapter 4 discusses the compliance checker. Chapter 5 presents an application case study. Chapter 6 discusses a series of scenarios and solutions. Chapter 7 concludes the thesis and draws out future work.

2.4 Distributed RBACs

2.4.1 PERMIS

The PERMIS (Privilege and Role Management Infrastructure Standards) project [11][12] is a role based access control infrastructure that is based on X.509 Attribute Certificates (AC). Attribute Certificate was first introduced by ANSI and standardised in the fourth edition of ISO/ITU-T X.509 Recommendation [60]. It is the certificate format of Privilege Management Infrastructure (PMI). PMI is similar to PKI (Public Key Infrastructure); the major difference is that PMI is to authorisation while PKI is to authentication. Public key certificate stores a user's name and the public key; an attribute certificate (AC) stores a user's name and privilege attributes assigned to him. In PMI, the issuer of the attribute certificate is called the Attribute Authority (AA). The root of trust of PMI is called the Source of Authority (SOA). SOAs may delegate their powers of authorisation to subordinate AAs. The AA has an attribute certificate revocation list (ACRL) that contains all the revoked attribute certificates. PMI and AC are to some extent similar to discretionary access controls (DAC), because the owner of the resource can grant users access right by issuing them attribute certificate.

PERMIS supports RBAC0. On the one side, permissions are stored in ACs and granted to roles. Permissions are the attributes contained in the AC, and the holder is the role. On the other side, roles are stored in ACs and granted to users. Roles are the attributes contained in the AC, and the holder is the user. PERMIS also supports RBAC1. By storing junior roles in the AC and assigning it to a senior role, roles are inherited in a hierarchical way. PERMIS supports delegation by appending an integer in the AC; the integer indicates the depth of permitted delegation.

The PERMIS architecture is consisted of a Privilege Allocator (PA) and privilege verification

system. The SOA and AA use PA to issue attribute certificates to users and sign the PMI policies. The SOA and AA store the roles as attributes in the certificate and put the AC into the publicly accessible LDAP (Light weight Deirectory Access Protocol) directory.

PERMIS has developed a Java version of the simplified Access Control Enforcement Function (AEF) and Access Control Decision Function (ADF). AEF and ADF are based on the Open Group standard AZN API [29] and the ISO Access Control Framework [30]. When a user accesses the resources, the AEF authenticates him. Then the ADF retrieves his role ACs, according to his LDAP DN (Distinguished Name), and the authorisation policy from the public LDAP directory. The authorisation decision is based on the requested action, the target resource, the permissions of the roles that the user holds and the policies. In PERMIS, Authentication is application specific and authorisation is application independent.

PERMIS has specified a policy language to define the policies. There several types of policies. The *subject* policy specifies the domains of the users; the domains are expressed as an LDAP subtree. The *SOA* policy lists the identities of the SOAs, usually in the form of the LDAP DNs. All the valid ACs must be signed by one of the SOAs, or one of the AAs that is delegated by an SOA. The *role hierarchy* policy specifies the role hierarchies within PMI. The roles are defined using type-value pairs as attributes; the types are identified by the globally unique object identifiers. The role hierarchy graph is a directed graph rather than a tree, because a role can have multiple superior roles and can also inherit from a set of subordinate roles, which cannot be represented by a tree. The *role assignment* policy specifies which roles can be assigned to which subjects by which SOAs. The depth of the delegation level of the role can be specified by an integer (particularly, integer zero means no delegation). Time constraints can also be applied to role assignment. The *target* policy specifies the target domains. The domains are distinguished by LDAP subtrees and are similar to subject domains. The *action* policy specifies all the valid actions that can be performed to the target resources. The action consists of a name and a number of arguments. The arguments will be passed into the PERMIS API by the Access Enforcement Function (AEF) at runtime. The *target access* policy consists of a series of target access clauses. Each clause contains a specific set of roles to perform the specified actions on the specified targets, only if the IF clause could be satisfied. The IF clause contains a series of conditions, a condition includes a comparison operator, an operand, and one or more variables/constants. PERMIS

supports user-defined operators.

PERMIS realizes a rigid distributed RBAC system. Roles are assigned to users as attributes in ACs by the issuing body. ACs are stored in public directory and can be retrieved by anyone. The deficiencies of PERMIS are that it cannot finely control the privileges assigned to users, and the issuing body must share the same definition of RBAC with the object domain which is sometimes unrealistic.

2.4.2 OASIS

OASIS stands for Open Architecture for Secure, Interworking Services. It is a role-based access control architecture that facilitates the interoperation between services in a distributed environment [2][3].

The administration of OASIS roles is intrinsically distributed. Each service maintains its own RBAC system. Users are not assigned privileges directly, instead they are assigned roles and roles are assigned privileges. Services interoperate via Service Level Agreements (SLA). An SLA is a contract between services; it contains role definitions and/or policy information. Services use SLA to interpret different RBACs. OASIS is integrated into an active, event-based middleware infrastructure. Any change of the environment is notified by a message, therefore role activation/deactivation are under supervision and can be reflected in real time [43].

Recognizing the limitations of delegation, OASIS builds privilege conveyance mechanism on its newly designed mechanism *appointment* to replace delegation. The central notion of appointment is that roles are able to grant roles, which are embedded in appointment certificates, to subjects without possessing the target roles, hence enabling subjects to activate more roles.

The activation of roles is controlled under role activation rules. A role activation rule specifies a series of conditions. The conditions include *prerequisite roles*, *appointment credentials* and *environmental constraints*. The conditions must be met in order for the corresponding role to be activated. A prerequisite role is the role that a subject must have acquired and has activated it before activating more roles. An example is that in order to access internal resources, users must prove that they are members of the role *internal users*. Some roles possess the ability to issue appointment credentials. With the appointment credentials, and maybe some other credentials required by the policy, subjects can activate more roles. The appointment credentials are

independent of the activation of the appointer role. The appointer does not have to explicitly possess the privileges contained in the appointment credential that he has issued. This is understandable because it is a quite common situation in real life. For example, the computer administrator in a hospital does not have the privilege to treat the patients, but he could grant the corresponding privileges to the patients' doctor. Environmental constraints include user independent constraints and user dependent parameters. For instance, user independent constraints can be the time of the day or IP address of the computer, user dependent parameters can be the name or position of the user.

OASIS roles are parameterised. This provides fine grained access control. For instance, patients can specify who may or may not see their medical records. A patient might express that "Clare may not see my medical record". OASIS parameters allow these side conditions to be identified and constraint checking permits such exceptions to be enforced.

Appointment has several advantages over delegation. First of all, privilege conveyance is totally under control. In order to complete a task, only those roles that are required during the process will be activated, therefore obey the principle of least privilege. Secondly, appointees will be assigned a different role from the appointer, which makes cascading delegation irrelevant. Thirdly, the appointer could grant privilege to appointee without possessing the privilege. This kind of situation is commonly viewed in real life but cannot be accomplished by delegation. Fourthly, delegation can be regarded as a special case of appointment, where the appointer can only grant a subset of roles that he owns to the appointee.

The revocation of appointment can be completed by three methods: by the appointer only; by anyone in the appointer role; by the rules of the system. Letting the appointer revoke the appointment seems to be a natural and straightforward way. But sometimes the appointer is unable to revoke, for instance, the appointer has retired or left. A solution is to allow anyone of the users who can activate the appointer role to make the revocation. A third method is that if a certain condition is met in the system then the appointment will be automatically revoked. These rules can be time, tasks and/or sessions. For instance, the appointment could be associated with a period of time; it will automatically be revoked after the expiry time. If the appointment is associated with some kind of task and the end time of the task is difficult to know in advance, then the revocation could be waken by the end of the task. It is especially helpful in a workflow environment. The

validation of the appointment can also be based on the appointer or appointee's session. If the session of the appointer/appointee ends, then the appointment automatically ends.

The precondition of OASIS interaction is that all the services are mutually trusted between each other. The authors also tried to establish a more common infrastructure to accommodate those previously unknown and untrusted services to interact. This remains to be an active research area.

2.4.3 Ponder

Ponder is a declarative, object-oriented policy language as a result of ten years of efforts [14]. It is aimed to specify security and management policy in distributed environment. It is a flexible, expressive and extensible policy language.

There are several kinds of policies. The authorisation policy defines what a principal of a subject domain could do on the targets of the object domain. Figure 2-1 is an example of authorisation policy. It says that the members of the NetworkAdmin domain are authorised to load, remove, enable or disable objects in the Nregion/switches domain.

```
Inst auth+ switchPolicyOps {
    Subject                 /NetworkAdmin;
    Target <PolicyT>        /Nregion/switches;
    Action                  load(), remove(), enable(), disable();
}
```

Figure 2-1 example of positive authorisation policy

The information filtering policy is used to modify the input/output parameters in an action. Its purpose is to restrict the information flow. Filters can only be applied to positive actions. Delegation policy defines the transfer level of access rights. It records the grantee that receives the delegated privileges. It can be associated with an authorisation policy, which contains the relevant subject, target and action. The refrain policy defines what actions the subjects cannot perform on the objects. It is similar to negative authorisation policy. The difference is that refrain policy is actively enforced by subjects because subjects might suspect the safety of the objects; negative

authorisation policy is passively enforced by the service controller. Obligation policy is event-triggered and defines the actions that must be performed by the subjects on the targets. Figure 2-2 is an obligation policy that specifies the user must be disabled after three consecutive login failures.

```
Inst oblig loginFailure {
        on                      3*loginfail(userid);
        subject                 s=/NRegion/SecAdmin;
        target <userT>          t=/NRegion/users ^ {userid};
        do                      t.disable() -> s.log(userid);
}
```

Figure 2-2 obligation policy

Constraint policy specifies the conditions under which the target policies are valid. A basic constraint policy is for single target policy and a meta policy is for a group of policies. Composite policy combines several related policies together to form a group policy, either by the same subject, target or other criteria. It facilitates policy management in large, complex enterprises. Role policy is a special case of group policy, in which all policies share the same subject. Roles can form hierarchies. They can have complex relationships between each other. For instance, a secretary role must mail a report to the manager role every Monday.

2.5 Policy based approaches

2.5.1 PolicyMaker and KeyNote

M. Blaze et al. summarized their work on trust management in [1]. The concept of trust management was first introduced in the PolicyMaker system [4] and was defined as "a unified approach to specifying and interpreting security policies, credentials, and relationships; it allows direct authorization of security-critical actions".

A trust management system has five basic components:

- Action, which is the operation that the subject performs on the object.
- Principal, which is the subject who has been legally granted some permission to perform

an action on the object.

- Policy, which defines the regulations that must be conformed to for the subjects to perform the actions.
- Credential, which allows subjects to delegate privileges to other subjects.
- Compliance checker, which generates an authorisation decision based on the given policy, a set of credentials, and a requested action.

There should be standard languages to describe the action, policy, and credential. The languages are shared by all the trust-management applications. The security configurations of the applications have exactly the same syntactic and semantic structure. Trust management unifies the notions of security policy, credentials, access control and authorisation. Unlike traditional certificates, which combine keys and names, trust management certificates combine keys and authorisations. The issuer delegates the capability to the holder through the certificate. Trust management systems are inherently extensible for distributed systems and versatile for many applications.

Trust management sums up distributed access control as "does the set *C* of credentials prove that the request *r* complies with the local security policy *P*?" [4]. The compliance checker takes in (*r*, *C*, *P*) and returns a result of compliance checking. The most important contribution of trust management is "a general purpose, application-independent algorithm for checking proofs of compliance" [4].

PolicyMaker was the first demonstration of trust management. It was introduced in [4] and its compliance-checking algorithm was later carried out in [20]. Credentials and policies of PolicyMaker are fully programmable and they are consisted of *assertions*. Assertions are two-value pairs in the form of (*f*, *s*), where *f* is a programme that describes the privileges and to whom they are being granted, *s* is the source of authority. In policies, *s* is always the preserved keyword *policy*, and in credentials, *s* is the public key of the issuer. Assertions can be written in any programming languages. The receiving end of the assertions must have an interpreter of the language in order to safely interpret them because credentials could be issued by untrusted authorities. A language named AWKWARD [4], which is a safe version of AWK [62], had been developed.

PolicyMaker does not assent to sticking on one particular language to write the assertions.

The advantage is that all the effort that "has gone into designing, analysing and implementing the PolicyMaker compliance-checking algorithm will not have to be redone every time an assertion language is changed or a new language is introduced." The proof of compliance and the language design are independent of each other and can be done separately.

The proof of compliance checking of PolicyMaker is consisted of a method of inter-assertion communication and a method for determining a result of success/failure. The inter-assertion communication is done via a write-only data structure. Firstly, a *blackboard* is created with only the request r on it. Then all the assertions (f_i, s_i) are run for one or more times. Each time an assertion runs it appends one or more records (i, s_i, R_{ij}) to the *blackboard*, where R_{ij} is an action that source s_i approves; f_i understands the meaning of R_{ij} but PolicyMaker does not. Finally, the *blackboard* will contain a record indicating the legitimateness of the request r. The authors had provided a mathematical formulation and proof of problem undecidability.

A limitation of PolicyMaker is that it only supports monotonic (non-negative) assertions.

KeyNote [5] was designed according to the same criteria but has made several improvements compared to its predecessor PolicyMaker. KeyNote system engine includes more functions than PolicyMaker and mitigates the burden of applications. KeyNote also requires that credentials and policy be written in a specific KeyNote language, which gains more efficiency and interoperability. The KeyNote assertion language is a simple and loops-restricted language; it has a minimal-sized interpreter.

Here is the basic structure of the assertion in KeyNote and an example:

<Assertion>:: <VersionField>? <AuthField> <LicenseesField>?
<LocalConstantsField>? <ConditionsField>?
<CommentField>? <SignatureField>?

Note: the notation "?" means zero or one repetition

Figure 2-3 KeyNote assertion structure

KeyNote-Version: 2
Local-Constants: Alice="DSA:4401ff92"
Bob="RSA:d1234f"

```
Authorizer: "RSA:abc123"
Licensees: Alice || Bob
Conditions: (app_domain == "RFC822-EMAIL") &&
            (address == "mab@keynote.research.att.com")
Signature: "RSA-SHA1:213354f9"
```

Figure 2-4 KeyNote assertion example

A KeyNote evaluator takes in a set of credentials, policies, requester public keys, an *action environment* and returns an application-defined string (usually authorised/denied) to the calling application. The *action environment* contains a set of attributes and is similar to the Unix shell environment.

KeyNote adopts a depth-first search (DFS) algorithm that recursively tries to satisfy both the *Conditions* field and the *Licensees* Key expression of at least one of the policy assertions. KeyNote's evaluation model is a subset of PolicyMaker's and is therefore guaranteed by the latter. At last, an assertion graph, which is composed of policy assertions and the issuer of the credentials, will be created to approve or deny the request.

KeyNote also has *non-negative credential* restriction as PolicyMaker.

Trust management models are based on rigorous delegation, where subjects cannot delegate more than they have; delegators and delegatees must be known and identified by their public keys.

2.5.2 XACML

XACML (eXtensible Access Control Markup Lauguage) is an OASIS standard that describes a general-purpose access control policy language and an access control decision request/response language in XML format [13][16].

The access control decision language lets the user ask whether a specific action is allowed or not by sending a message and receiving the response. The response contains one of the following four values:

- Permit. The action is allowed.
- Deny. The action is not allowed.
- Indeterminate. An error occurred or more information is required to make a decision.

- Not Applicable. The service is unable to answer the request.

A user makes a request to a Policy Enforcement Point (PEP). The PEP sends the user's attributes, the resource's attributes, the requested action, together with some other relevant information to a Policy Decision Point (PDP). The PDP checks the input against the relevant policy and returns the answer to PEP. PEP will allow or deny the request based on the decision.

The policy is written in XACML policy language. Each policy document contains one Policy/PolicySet root element. A Policy/PolicySet contains several Rules or other Policy/PolicySet. To make an access control decision, the PDP first locates the Targets that apply to the request, evaluates each Rule related to the Target. Then a PEP or PDP extracts attributes from the request, the resources and the environment through *AttributeDesignator* or *AttributeSelector*. Multiple attributes are stored in Bags. The PEP/PDP use system or user-defined functions to compare the attributes according to the Rules and return a result. The final decision is made according to the combined result of all the rules via Policy Combining Algorithms or Rule Combining Algorithms.

The drawbacks of XACML are that it does not integrate RBAC, users have to define and include their own RBAC; also it does not support delegation, which constrains scalability of the system.

2.6 Summary

Table 2-1 summary of related work on distributed access control

	Support RBAC	Decentralisation mechanism	Fine-grained access control	Comment
PERMIS	Yes	Uniform RBAC is administered in different locations	No	Roles are stored in the certificate as attributes
OASIS	Yes	Through Appointment	Partially	Uses appointment to replace delegation

PolicyMaker and KeyNote	No	Through rigid delegation	Partially. Access rights are directly assigned to subjects	drew the concept of trust management
XACML	No	No	Yes. Via attributes of both requester and resource	OASIS standard
Ponder	Yes	Through domain policy and delegation policy	Partially	

As we can see from the above table, most mechanisms adopt delegation as the solution of decentralised management of trust. Delegation is restricted because it requires that the identities of the delegator and delegatee must be known in advance. OASIS introduces a new decentralised mechanism called appointment. More research is needed in this area.

3 Trust and the Mojoy Trust Policy Language

3.1 Foundation of trust

The concept of trust has been addressed within many disciplines, including philosophy, psychology, sociology, transaction economics and organization theory. It has widely acknowledged that trust is complex and multidimensional [10][27].

In this thesis, we discuss trust in a distributed computing environment. Trust can be generic description of the specific subject. Trust can be particular privileges of a subject. Trust can be transferred from one subject to another. Service providers trust subjects by allowing them to access protected services and/or resources.

3.1.1 Initialization of trust

There are several ways for a subject to start its trust relation with the unknown world. Whatsoever, the subject has to trust some other subjects unconditionally at the very beginning.

Recommendation

In real life, recommendation is the most common method to help making a decision. Recommendation is usually obtained from someone familiar, or the media, newspaper, etc. The credibility of recommendation highly depends on the source. It could be unreliable because it is very subjective.

Reputation

Reputation is used to establish trust between unfamiliar parties. It does not require prior contact experience with each other. It is a collective opinion of the public about an unknown party. Therefore it is more reliable than recommendation. But it is vulnerable to collusion and can be deliberately manipulated.

Experience

Trust is closely related to previous experiences because experiences can be good evaluation criteria to predict the outcome of future interactions. Experiences may be consisted of vague memory or concrete records of transaction history. Depending on the knowledge learned from past experiences, the level of trust may increase or decrease. Experiences can also be gained and shared by some other trusted parties. In this case, it is similar to recommendation.

Root of trust

An entity needs to first trust at least one subject in order to establish trust with more subjects. That particular subject(s) is called *root of trust*. The *root of trust* is recognized by its identity. This trust relationship is usually unconditional and uncaused. For instance, the administrator has full power over the whole system, whoever he/she is. The subject trusts the *root of trust* only to a limited scope.

3.1.2 Conveyance of trust

Trust can be transferred from one subject to another. It can be disclosed to third parties. This facilitates a subject to establish trust with a previously unknown subject through some already familiar subjects.

Subjects are distinguished by public/private key pairs. A public key is associated with only one private key and vice versa. The probability that two subjects having the same public/private key pair is so tiny that it is negligible. The public key can be publicised to everyone so that anyone (even those who are totally untrusted strangers) can know the public key and the owner. The private key is kept secret. A subject proves his/her ownership of the public key by successfully decrypting/encrypting a particular message using the private key. A subject can have multiple public/private key pairs. This means those systems that require users to be identified by their identities cannot merely depend on public/private keys; there must be some extra methods to associate the identity with the public/private key, for instance, requiring ID card before allocating public/private key.

Trust can be defined as properties. A property can be a generic statement of a subject, such as

attributes. The meaning of the statement is understood by the receiving subject, the interpretation accords to some commonly accepted rules. For instance, Alice is a doctor in a hospital. She is approved by the hospital. When she goes to another hospital, the new hospital finds out that she is a doctor in the previous hospital and grants her corresponding privileges according to her former position. It is very likely that Alice does not have the same responsibilities in those two hospitals. The common rule negotiated by the two parties to interpret the roles is known as Service Level Agreement (SLA) [17]. SLA is a bilateral contract that specifies the role definitions. With SLA, roles can be created remotely according to the same definition and be revoked synchronously in real time. A property can also be a particular capability. The capability precisely describes what the subject can do on the specific targets. The meaning of the capability is clearly defined by the original authority and cannot be misinterpreted. For instance, a capability could be "Alice can read the public resource in hospital A". This capability must be interpreted uniformly at any location. Alice cannot have more privilege than the capability.

Properties can be owned, obtained and/or controlled. The subject who owns the properties is the owner. The owner can grant the properties to any trusted subjects, i.e. grantees. The grantees therefore obtain the properties from the owner. The original owner decides whether the grantees can further grant the properties to other subjects. The subject who controls the properties is the controller. The controllers do not own the properties and thereby cannot perform the actions indicated by the properties. But they can grant the properties to other subjects therefore those subjects could own the properties and legally perform the indicated actions.

A subject grants properties to another subject in the form of electronic credentials, a.k.a. certificate. For instance, X.509 certificate is a widely recognized certificate format. The granting subject is the certificate issuer and the receiving subject is the certificate holder. Properties are stored in the certificate. The issuer digitally signs the certificate. The digital signature provides authenticity, integrity and non-repudiation. The receiving party of the certificate can verify that the certificate is from the authentic issuer, the content of the certificate has not been tampered, and the subject is the true holder. Integrity can be verified by recalculating and matching the message digest value of the certificate. The public key of the issuer must be known in advance to verify the authenticity of the certificate. It is done by re-computing the digital signature using the issuer's public key. With a digital signature, the issuer cannot deny the issuance of the certificate, because

no one could sign the same signature without owning the specific private key. The holder of the certificate could be verified by successfully encrypting/decrypting a random message.

The issuer uses certificate to convey his/her trust to the certificate holder. The holder uses the certificate to prove his/her credibility to third parties. The certificate could be disclosed to anyone. They are tamperproof. The propagator of the certificate could be anyone other than the issuer/holder. Certificate can be used to justify the validity of other credentials. For instance, Alice is a doctor. She has a credential from the hospital to prove her identity and position. The hospital is further certified by the National Health Service (NHS). In this case, Alice can present the hospital's certificate, which is issued by NHS, to help confirm her identity and position. A trust chain is created from NHS to Alice.

3.1.3 Decentralised administration of trust

In a highly distributed computing environment, the administration of trust should be disseminated to many subjects who are not the original owner/controller of trust. The mostly commonly adopted decentralisation mechanism is delegation. The owner of the properties grants a subset of the properties to another subject, the owner is known as the grantor/delegator, the receiver is known as grantee/delegatee. The grantee can further grant the properties to other subjects. Delegation level (i.e. how far the properties can be transferred) is controlled by the delegator.

However, sometimes the subject does not always own the properties before delegation. For instance, the system administrator can create new users and assign relevant privileges to specific users. The users can execute system operations, but the administrator does not have the privilege to execute system operations.

We have designed a new decentralised administration mechanism, trust empowerment. The service provider defines the trust regulation in its local policy. In the policy, the server specifies what subjects own/control what properties, whether the properties can be further granted to other subjects and how far they can be transferred. The subjects can be specified either by their identities or properties. Different server keeps its private personal policy. The same subject trusted at one server may not be trusted at another.

3.1.4 Variation of trust

Trust is dynamic. It changes as environment varies. It evolves as knowledge and experiences accumulate. Previously trusted subjects may become untrusted. For instance, trust is limited by time constraint. A certificate is only valid within its validity time period. After that period, the certificate will become invalid. Furthermore, the certificate could be revoked before its expiry time. A consequence of this annulment is that all derived trust depending on the trusted properties of the certificate will become invalid. Previously untrusted subjects may become trusted. For instance, the subject provides extra evidence to satisfy the security criteria. The credibility of the subject could be affected by outside changes. For instance, the common rule, SLA (Service Level Agreement), used to interpret the roles changes. As a consequence, the subject loses the privileges to access the resource.

Service providers set rules in their local policies to filter requests. The rules describe the conditions that the subjects must meet in order to be trusted and the privileges they can obtain once become trusted. Conditions could consist of subjects' identities, attributes, capabilities and/or environmental parameters. Privileges could consist of attributes, capabilities and controls. Conditions and privileges are many-to-many relationship.

Trust could evolve. As the server's experience with the subject accrues, the trust level of the subject could increase/decrease. The same subject with the same properties under the same policy could gain/lose privileges according to the accrued experience.

3.2 Trust policy language

Sandhu engineered the structure of security into four layers. From top to bottom, they are policy, model, architecture and mechanism [33]. Policy is the high-level organizational requirements and mechanism is the implementation of the security design.

M. Blaze et al. proposed a *trust management* infrastructure to solve the trust issues in the distributed systems. It contains the following elements [4]:

- Certificates, a.k.a. electronic credentials. They are used to transfer trust information between entities.
- Policy, which is stored and trusted locally. It expresses the local security regulation,

trusted authorities, trusted relationships, etc.

- Compliance checker, which is a computer programme that takes in user certificates and local policy, examines them, and generates an access decision, which usually is *grant* or *deny*.

Policies and certificates are written in a language that can be understood by all the entities involved in the system. The infrastructure of *Trust management* has been widely accepted and is considered to be a correct direction.

Recognizing the complex nature of trust, we develop a policy language to describe the basic entities and their trust relationships. We present the syntax, semantics and an implementation of the proposed policy language in this chapter. The syntax is expressed in BNF (Backus Naur Form) [61]. The trust policy language could be implemented via various ways. XML is a good format to express the policy language; it has been adopted by PERMIS [12], XACML [13] and Akenti [42]. The advantages of using XML are:

- XML is human readable, easy to maintain and platform independent.
- The syntax of XML document can be validated against a schema/DTD file, which could help reduce format errors.
- There are a number of handy tools and libraries to operate XML document.

An XML schema file will be provided as the definition of the policy language.

3.2.1 Why need a policy language

There are a number of ways to specify, represent and administer policy [17]. Obviously, natural languages are the best choice but they are inappropriate because of their ambiguous nature. Efforts have been made to visualise security policies [37][38]. Some components such as role-privilege mapping and hierarchies are straightforward and can be easily done. But some other areas such as constraint and conflict are still difficult to visualise. Many RBAC models use formal, logic-based languages because they are suited for formal reasoning, i.e. the semantics and syntax can be formally checked and analysed by a programme.

According to our requirements, policy is used to express our new trust model of *trust empowerment*, describe the trust relationships between different entities, support various RBAC models, and implement fine-grained access control. Access matrix, ACL and database are not

expressive enough. The ideal choice is a logic-based language which is simple, easy and expressive.

A policy language is not a programming language. It does not have to have loop/sequence clauses. It has subjects, objects and actions as the basic elements. It describes the conditions under which the subjects are trusted. It associates the conditions with privileges that the subjects could possess. In a nutshell, the policy language describes the trust relationships between subjects.

Advantages and disadvantages

The advantages of policy based access control over other approaches have been discussed in [39][40], the conclusion is that policy based access control is the best choice for distributed environment. Policy is separated from the application. It is independent of the application and can be updated dynamically to reflect frequent security requirement changes. Policy is portable and can be reused between different applications. The enforcement is automatically carried out via the programme. Policies are powerful enough to express complex situations such as fine-grained access control, environmental constraints, advanced RBAC models and intricate trust relationships. Policies can be distributed and uniformly enforced. These advantages clearly reveal that policy based mechanism is ideal for distributed access control.

Policy based access control has its disadvantages. All the systems involved must be able to understand the language which is sometimes difficult to accomplish in a widely distributed environment. Different organizations have different requirements, some require a simple language and others require a complex one, an all-purpose language is hard to design. A compliance checker is needed to process the certificates and policies. Of course the compliance checker should be able to understand all the possible formats, which is very difficult, if possible, to achieve.

3.2.2 Subject

Syntax:

subjects ::= subject { subject } | **any_subject**

subject ::= public_key

public_key ::= **string**

issuers ::= subjects

holders ::= subjects

Description:

A *subject* is an entity that performs actions in a system; it can be a user, a program or a server. A *subject* is distinguished by its *public key*. The *public key* is computed by a certain algorithm and specific parameters. The *public key* is encoded into a readable string through an algorithm such as BASE64. *Subjects* is a set that consists of one or more *subject* or *any subject*. *Any subject* denotes any legal and possible *subject*. *Issuers* and *holders* are special *subjects*. *Issuers* sign the certificate and *holders* own the certificate. A *subject* proves its authenticity by providing evidence of owning the corresponding private key. This could be done by several ways, such as encrypting/decrypting a randomly generated message. Whether the same *subject* could hold multiple public/private key pairs is not of our concern.

Implementation:

```
<xs:element name="Subjects">
    <xs:complexType>
        <xs:choice>
            <xs:element ref="Subject" maxOccurs="unbounded"/>
            <xs:element ref="AnySubject"/>
        </xs:choice>
    </xs:complexType>
</xs:element>
<xs:element name="Subject">
    <xs:complexType>
        <xs:sequence>
            <xs:element ref="PublicKey"/>
        </xs:sequence>
    </xs:complexType>
</xs:element>
<xs:element name="PublicKey" type="xs:string"/>
<xs:element name="AnySubject"/>
<xs:element name="Issuers">
```

```
        <xs:complexType>
            <xs:sequence>
                <xs:element ref="Subject" maxOccurs="unbounded"/>
            </xs:sequence>
        </xs:complexType>
    </xs:element>
    <xs:element name="Holders">
        <xs:complexType>
            <xs:sequence>
                <xs:element ref="Subject" maxOccurs="unbounded"/>
            </xs:sequence>
        </xs:complexType>
    </xs:element>
```

3.2.3 Attribute

Syntax:

attributes ::= attribute { attribute } | **any_attribute**

attribute ::= name value

name ::= **string**

value ::= **string**

Description:

Attribute is generic description of a subject. An *attribute* is a *name* and *value* pair. The *name* and *value* are of string types. For instance, "Alice is a student" could be represented as "name=Alice, position=student". A role could also be stored as an *attribute*, for example, "role=user". The meaning of the *attribute* is interpreted by the object application. *Attributes* is a set that consists of one or more *attribute* or *any attribute*. *Any attribute* denotes any legal and possible *attribute*.

Implementation:

```
    <xs:element name="Attributes">
        <xs:complexType>
```

```
            <xs:choice>
                <xs:element ref="Attribute" maxOccurs="unbounded"/>
                <xs:element name="AnyAttribute"/>
            </xs:choice>
        </xs:complexType>
    </xs:element>
    <xs:element name="Attribute">
        <xs:complexType>
            <xs:sequence>
                <xs:element name="Name" type="xs:string"/>
                <xs:element name="Value" type="xs:string"/>
            </xs:sequence>
        </xs:complexType>
    </xs:element>
```

3.2.4 Capability

Syntax:

capabilities ::= capability { capability }

capability ::= targets actions

targets ::= target { target } | **any_target**

actions ::= action { action } | **any_action**

target ::= **string**

action ::= **string**

Description:

Capabilities is a set that consists of one or more *capability*. A *capability* is what a subject can do on specific *target*. A *capability* consists of *targets* and *actions*. *Targets* consists of one or more *target* or *any target*. In the current version, a *target* is a string. In the future, we are going to define *target* in Uniform Resource Identifier (URI) format. URI is a string of characters for identifying an abstract or physical resource [52]. URI can be finely classified into Uniform Resource Locator (URL) and Uniform Resource Name (URN) [51]. The former represents a resource by its current

location and access method. The latter represents a resource by its globally unique name and can be persistent even if the resource becomes unavailable. *Any target* denotes any legal and possible *target*. *Actions* consists of one or more *action* or *any action*. An *action* is application specific and must be understood by the object application. It is usually encoded into a readable string. *Any action* denotes any legal and possible *action*.

Implementation:

```
<xs:element name="Capabilities">
    <xs:complexType>
        <xs:sequence>
            <xs:element ref="Capability" maxOccurs="unbounded"/>
        </xs: sequence >
    </xs:complexType>
</xs:element>
<xs:element name="Capability">
    <xs:complexType>
        <xs:sequence>
            <xs:element ref="Targets">
            <xs:element ref="Actions">
        </xs:sequence>
    </xs:complexType>
</xs:element>
<xs:element name="Targets">
    <xs:complexType>
        <xs:choice>
            <xs:element ref="Target" maxOccurs="unbounded"/>
            <xs:element ref="AnyTarget"/>
        </xs:choice>
    </xs:complexType>
</xs:element>
<xs:element name="Actions">
```

```
        <xs:complexType>
            <xs:choice>
                <xs:element ref="Action" maxOccurs="unbounded"/>
                <xs:element ref="AnyAction"/>
            </xs:choice>
        </xs:complexType>
    </xs:element>
    <xs:element name="Target" type="xs:string"/>
    <xs:element name="AnyTarget"/>
    <xs:element name="Action" type="xs:string"/>
    <xs:element name="AnyAction"/>
```

3.2.5 Control

Syntax :

controls ::= control { control }

control ::= attributes | capabilities

Description :

Controls is a set that consists of one or more *control*. *Control* is a set that consists of *attributes* and/or *capabilities*. Attributes and capabilities are also called properties. Sometimes a *subject* does not own the properties; instead it has *control* over the properties. This means that the controlling *subject* can grant the properties to other *subjects* thereby the grantees will own the properties and can legally perform the specified actions, but the controlling *subject* cannot perform the same actions. *Attributes* and *capabilities* are defined in previous sections.

Implementation :

```
    <xs:element name="Controls">
        <xs:complexType>
            <xs:sequence>
                <xs:element ref="Control" maxOccurs="unbounded"/>
            </xs:sequence>
        </xs:complexType>
```

```
</xs:element>
<xs:element name="Control">
    <xs:complexType>
        <xs:all>
            <xs:element ref="Attributes"/>
            <xs:element ref="Capabilities"/>
        </xs:all>
    </xs:complexType>
</xs:element>
```

3.2.6 Constraint

Syntax :

constraints ::= constraint { constraint }

constraint ::= time_constraint | ip_constraint

time_constraint ::= start_time end_time

start_time ::= **string**

end_time ::= **string**

ip_constraint ::= **string**

Description :

Constraints is a set that consists of one or more *constraint*. A *constraint* contains environmental parameters, which are in contrast to users' attributes and are independent of users. A *constraint* consists of one *time constraint* or one *IP constraint*, or a combination of them. A *time constraint* consists of one *start time* and one *end time*. *Start time* and *end time* are expressed in the form of a string, for instance, "16/10/2004 20:06:00" or "09:00:00". An *IP constraint* is an Internet IP address expressed in the form of a string, for instance, "129.234.198.1" or an IP address segment "192.168.0.1/24".

Implementation :

```
<xs:element name="Constraints">
    <xs:complexType>
        <xs:sequence>
```

```
                <xs:element ref="Constraint" maxOccurs="unbounded"/>
            </xs:sequence>
        </xs:complexType>
    </xs:element>
    <xs:element name="Constraint">
        <xs:complexType>
            <xs:sequence>
                <xs:element ref="TimeConstraint" minOccurs="0"/>
                <xs:element ref="IPConstraint" minOccurs="0"/>
            </xs:sequence>
        </xs:complexType>
    </xs:element>
    <xs:element name="TimeConstraint">
        <xs:complexType>
            <xs:sequence>
                <xs:element name="StartTime" type="xs:string"/>
                <xs:element name="EndTime" type="xs:string"/>
            </xs:sequence>
        </xs:complexType>
    </xs:element>
    <xs:element name="IPConstraint" type="xs:string"/>
```

3.2.7 Condition

Syntax :

conditions ::= condition { condition }

condition ::= [subjects] [issuers] [holders] [attributes] [capabilities] [constraints]

Description:

Conditions is a set that consists of one or more *condition*. *Condition* consists of *subjects*, *attributes*, *capabilities*, *constraints*, or any combination of them. Satisfying the *conditions* set means satisfying all the *condition*. *Subjects*, *issuers*, *holders*, *attributes*, *capabilities* and

constraints are defined in previous sections.

A *conditions* set is associated with one or more *privileges* sets. A *subject* must satisfy at least one *conditions* set in order to be trusted. Trusted *subjects* can be granted the *privileges* contained in the corresponding *privileges* set. See following sections for more details.

Implementation:

```
<xs:element name="Conditions">
    <xs:complexType>
        <xs:sequence>
            <xs:element ref="Condition" maxOccurs="unbounded"/>
        </xs:sequence>
    </xs:complexType>
</xs:element>
<xs:element name="Condition">
    <xs:complexType>
        <xs:all>
            <xs:element ref="Subjects" minOccurs="0"/>
            <xs:element ref="Issuers" minOccurs="0"/>
            <xs:element ref="Holders" minOccurs="0"/>
            <xs:element ref="Attributes" minOccurs="0"/>
            <xs:element ref="Capabilities" minOccurs="0"/>
            <xs:element ref="Constraints" minOccurs="0"/>
        </xs:all>
    </xs:complexType>
</xs:element>
```

3.2.8 Privilege

Syntax:

privileges ::= privilege { privilege }

privilege ::= attributes | capabilities | controls

Description:

Privileges is a set that consists of one or more *privilege*. A *privilege* is what a *subject* can do in a system. A *privilege* consists of *attributes*, *capabilities*, *controls*, or any combination of them. One *privileges* set is associated with one or more *conditions* sets. If a *subject* satisfies the associated *conditions* set, then it gains the *privilege* contained in the *privileges* set. *Attributes*, *capabilities* and *controls* are defined in previous sections.

Implementation:

```
<xs:element name="Privileges">
    <xs:complexType>
        <xs:sequence>
            <xs:element ref="Privilege" maxOccurs="unbounded"/>
        </xs:sequence>
    </xs:complexType>
</xs:element>
<xs:element name="Privilege">
    <xs:complexType>
        <xs:all>
            <xs:element ref="Attributes"/>
            <xs:element ref="Capabilities"/>
            <xs:element ref="Controls"/>
        </xs:all>
    </xs:complexType>
</xs:element>
```

3.2.9 Rule

Syntax:

rules ::= rule { rule }

rule ::= conditions privileges

Description:

Rules is a set that consists of one or more *rule*. A *rule* specifies under what condition and to what extent a *subject* can be trusted. A *rule* consists of *conditions* and *privileges*. If a subject

satisfies the *conditions* set, it acquires all the *privilege* in the *privileges* set in the same *rule*. A *subject* could satisfy more than one *rule* at the same time and thereby gains as many *privileges* as possible. *Conditions* and *privileges* are defined in previous sections.

Implementation:

```
<xs:element name="Rules">
    <xs:complexType>
        <xs:sequence>
            <xs:element ref="Rule" maxOccurs="unbounded"/>
        </xs:sequence>
    </xs:complexType>
</xs:element>
<xs:element name="Rule">
    <xs:complexType>
        <xs:sequence>
            <xs:element ref="Conditions"/>
            <xs:element ref="Privileges"/>
        </xs:sequence>
    </xs:complexType>
</xs:element>
```

3.2.10 Policy

Syntax:

policy ::= version rules [conditions] [privileges]

version ::= **XML name space**

Description:

The server specifies its security regulation in the form of *policy*. *Policy* consists of *version*, *rules*, *conditions* and *privileges*. *Conditions* and *privileges* are optional; they are used to define common conditions and privileges in order to simplify the structure of the document. Different versions of the policies are distinguished by *version*, which is XML name space. The current version of our implementation is *http://www.dur.ac.uk/chenxi.huang/mojoy0.1*. *Conditions* and

privileges are defined in previous sections.

Implementation:

```
<xs:element name="Policy">
    <xs:complexType>
        <xs:all>
            <xs:element ref="Rules"/>
            <xs:element ref="Conditions" minOccurs="0"/>
            <xs:element ref="Privileges" minOccurs="0"/>
        </xs:all>
    </xs:complexType>
</xs:element>
```

3.3 Certificates

Certificates are used to transfer trust from one subject to another. Trust is defined as a set of properties and stored in the certificate. The subject that grants the trust is the certificate issuer and the subject that receives the trust is the certificate holder. The issuer digitally signs the certificate. The digital signature guarantees the certificate's integrity, authenticity and non-repudiation.

The certificate shares part of the syntax and semantics of the policy. But there are several visible differences.

- The certificate contains issuer(s) and holder(s); the policy does not.
- The certificate is digitally signed by the issuer(s) and can be disclosed to anyone; the policy does not bear a digital signature because it is locally trusted, private and cannot be divulged.
- The certificate is restricted by some particular conditions contained in the certificate, for instance, validity period; a policy is not limited by a constraint.

Syntax:

certificate ::= version holders [attributes] [capabilities] [controls] conditions issuers signature

signature ::= **string**

Description:

The certificate consists of *version*, *holders*, *attributes*, *capabilities*, *controls*, *conditions*, *issuers* and *signature*. *Attributes*, *capabilities* and *controls* are optional. Trust is defined as sets of *attributes*, *capabilities* and *controls*. *Conditions* restricts the validity of the certificate. *Holders* are the subjects who own the trust. *Issuers* are the subjects who grant the trust to the *holders* and digitally sign the certificate. The *signature* is a message digest value of the certificate encrypted by the issuer's public key, which is usually encoded into a readable string. *Version*, *holders*, *attributes*, *capabilities*, *controls*, *conditions* and *issuers* are defined in previous sections.

Implementation:

```
<xs:element name="Certificate">
    <xs:complexType>
        <xs:all>
            <xs:element ref="Holders"/>
            <xs:element ref="Attributes" minOccurs="0"/>
            <xs:element ref="Capabilities" minOccurs="0"/>
            <xs:element ref="Controls" minOccurs="0"/>
            <xs:element ref="Conditions"/>
            <xs:element ref="Issuers"/>
            <xs:element ref="Signature"/>
        </xs:all>
    </xs:complexType>
</xs:element>
<xs:element name="Signature" type="xs:string"/>
```

3.4 Summary

In this chapter, we have discussed the foundation of trust. Trust is established via recommendation, reputation and experience. The very first trusted subject is called the *root of trust*. Trust can be defined as a set of properties, stored in a certificate and distributed to anyone. Trust varies; it can increase/decrease. Most of the traditional mechanisms adopt delegation as the decentralised administration mechanism of trust. We have proposed a new mechanism, *trust empowerment*, to overcome the limitations of delegation. The new mechanism is implemented

through a policy language and a compliance checker. We have provided the syntax, semantics and an XML implementation of the policy language. The XML schema of both the policy and certificate is given.

4 The Compliance Checker

The function of the compliance checker is to check the validity of the certificates, look through the local trust policy, find out whether the request meets the policy and could be supported by the certificates. The inputs are user's request, user's certificates, local trust policy and environmental parameters. The output is a Boolean value indicates whether the request is permitted or not. Additional information is also generated, for instance, the conditions that are not satisfied and/or extra supportive documents are required, etc.

4.1 Java and XML

The compliance checker is written in Java and the policy and certificate are written in XML. We need to examine the available programming tools for Java and XML. There are two popular APIs (Application Programming Interface) available to process XML documents. They are SAX (Simple API for XML) and DOM (Document Object Model). SAX provides an event-based framework for parsing XML data, which is the process of reading through the document and breaking down the data into usable parts. SAX defines all the possible events that could happen during the parsing procedure. For example, SAX defines an org.xml.sax.ContentHandler interface that defines *startDocument()*, *startElement(...)*, *error(...)* and *warning(...)*. Implementing this interface allows complete control over these portions of the XML parsing process. A set of errors and warnings is defined, allowing handling of unexpected situations that occur during parsing, such as invalid or not well-formed document. DOM provides a representation of an XML document as a tree. Traversal and manipulation of tree structures are easy to accomplish in programming languages. DOM reads an entire XML document into memory, stores all the data in nodes, so the entire document is very fast to access.

SAX and DOM are both programming language independent. The significant drawback of DOM is that it consumes a lot of resources. Because DOM reads an entire document into memory, application could be slowed down or even crashed. The larger and more complex the document, the more pronounced this performance degradation becomes.

JAXP is Sun's Java API for XML Parsing. JAXP does not compete with or replace either of SAX and DOM, it does add some convenience methods to try to make the XML APIs easier to use for Java developers. It conforms to the SAX and DOM specifications. It does not redefine SAX or DOM behaviour, but ensures that all XML-conformant parsers can be accessed within Java application through a standard pluggable layer.

These above APIs should be distinguished from XML parser. Actually, they provide framework for parsers to use. A parser must be supplied to SAX and DOM to perform any XML parsing. There are many excellent parsers available in Java, such as Apache Xerces, Oracle's XML Parser and IBM's XML4J.

JDOM is designed to the 80/20 rule of usability, i.e., for the 80% of the time we use 20% of the functions. It attempts to solve the deficiencies widely recognized in SAX, DOM, and JAXP. It seeks to provide a Java-centric, high-performance alternative in most cases. It is not based on DOM or SAX, but rather allows a user to deal with an XML document in tree form without the idiosyncrasies of DOM. At the same time, it provides the high performance of SAX, allowing very quick parsing and output. Additionally, it is namespace-aware; it supports validation through DTDs and XML Schema.

Considering the nature of our implementation, the compliance checker needs to traverse the XML document to find a match entry, but does not need to create, modify, insert or append anything into the existing document; the structure of the credential and policy is already known, the decision should be made as quick as possible. DOM is not adequate for the job and SAX is too complicated to deal with some of the common operations such as retrieving a specific element text according to a given XPath. We do not have cross-language concerns and Java itself is platform independent, JDOM is the ideal choice for the compliance checker. Actually, we have saved more than 60% programming time after adopting JDOM.

4.2 Prototype Interfaces

Three major interfaces have been defined for the compliance checker and they all start with a capital *I*. They are *IComplianceChecker*, *ICertificate* and *IPolicy*. These interfaces construct the foundation of the compliance checker and delineate the basic methods. Different software vendors could provide different implementation of the compliance checker. Application developers would

not have to worry about the compatibility issues as long as the implementations comply with the same specification. Developers can replace the underlying implementation without affecting super-stratum applications. Figure 4-1 is the UML graphs for the classes. From now on, interface, class, method definitions and variable declarations are utterly described in UML and our programs are written in Java.

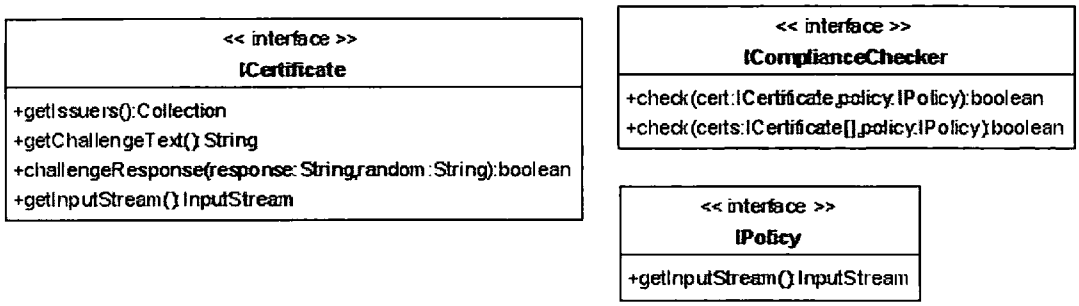

Figure 4-1 the interfaces of the compliance checker

The interface *ICertificate* defines the basic methods of a certificate. The method *getIssuers():Collection* returns the issuer(s) of the certificate in a *Collection*. The method *getChallengeText():String* returns a randomly generated text to test the ownership of the alleged certificate holder. The method *challengeResponse(random:String, response:String):boolean* tests whether the challenge has been successfully digitally signed by the private key of the certificate. The method *getInputStream():InputStream* returns the source of the certificate.

The interface *IPolicy* defines the methods of the local trust policy. Unlike certificate, the local trust policy is stored in a secure place and does not have to be signed. It has only one method *getInputStream():InputStream* which returns the XML source of the policy.

The interface *IComplianceChecker* defines the methods of the compliance checker. It has two methods. The method *check(cert:ICertificate, policy:IPolicy):boolean* takes in two parameters, *cert* is a certificate, *policy* is the local trust policy. It checks whether the certificate complies with the trust policy, if so, it returns true, otherwise returns false. It throws an exception *CheckerException* upon error, for example, the format error of the certificate or policy. The method *check(certs:ICertificate[], policy:IPolicy):boolean* takes in two parameters, *certs* is collection of certificates, *policy* is the local trust policy. The method checks whether the set of certificates comply with the local trust policy. If so, it returns true, otherwise false. Furthermore, this method tries to find a trust train between the first certificate and the policy rules through the

rest of the certificates. It throws an exception *CheckerException* upon error.

4.3 Implementation

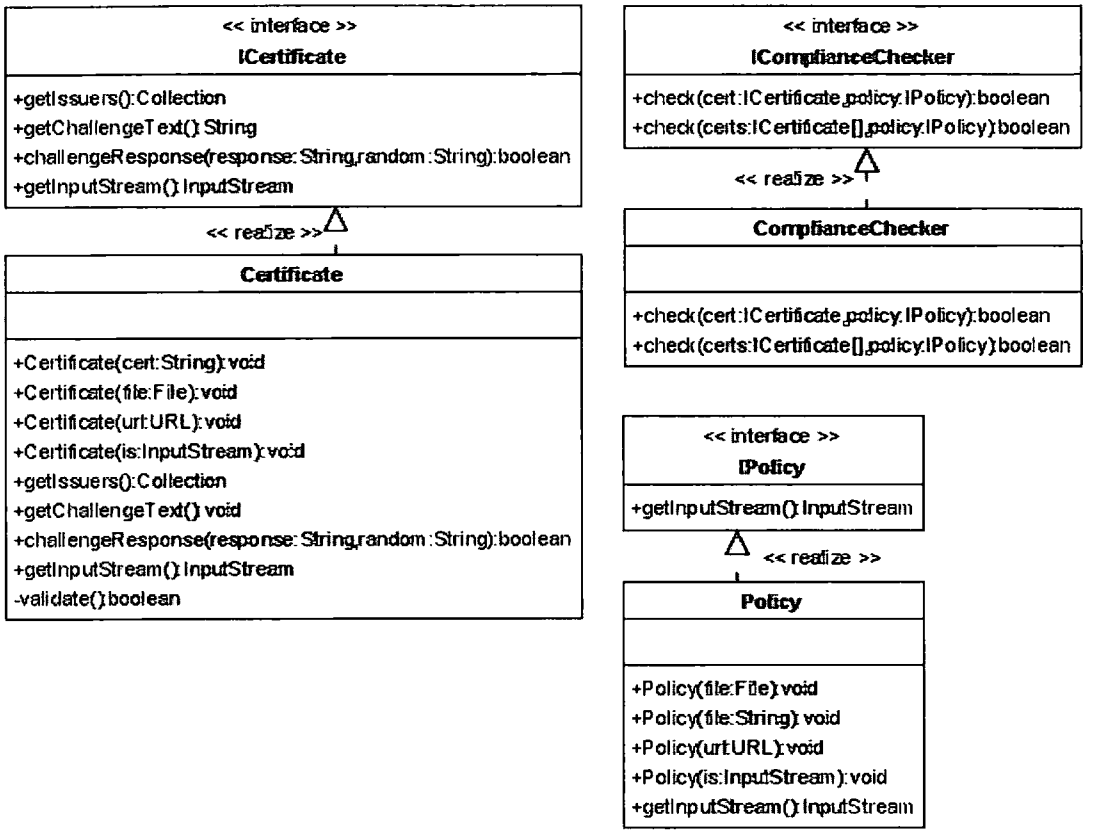

Figure 4-2 the Java implementation of the compliance checker

Figure 4-2 is the UML diagram of our implementation of the compliance checker. Class *Certificate* is the implementation of *ICertificate*. Despite the four methods defined by the super interface, it has defined three more methods. *Certificate(cert:String)* is a constructor method. It takes in one parameter *cert*, which is the file name of the certificate. It throws *CertException* in case of file format or open error. The method *validate():boolean* tests the validity of the certificate, including signature, time validity and revocation test. It returns true if the validation process passes otherwise false. Class *Policy* is the implementation of *IPolicy*. Class *ComplianceChecker* is the implementation of *IComplianceChecker*.

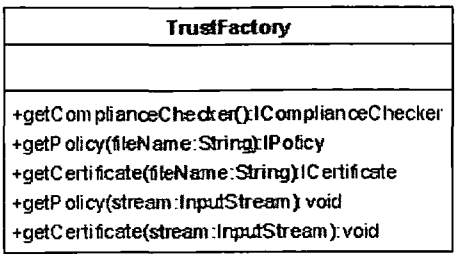

Figure 4-3 the instance factory

Class *TrustFactory* is the instance factory. It automatically selects the best version of the implementation, generates and returns the instances of the interfaces. It has the following methods:

getCertificate(cert:String):ICertificate and *getCertificate(cert:InputStream):ICertificate*

are the factories of the interface of *ICertificate*. They take in different parameters and return an instance of *ICertificate*.

getPolicy(policy:String):IPolicy and *getPolicy(policy:InputStream):IPolicy*

are the factories of the interface *IPolicy*. They take in different parameters and return an instance of the interface *IPolicy*.

The method *getComplianceChecker():IComplianceChecker* returns an instance of the interface *IComplianceChecker*.

The compliance checking procedure is shown in Figure 4-4.

1. First of all, it checks the validity of the local trust policy. If this step fails, possibly caused by ill-formed XML format or unrecognised policy version, the checking procedure fails and an Exception will be thrown.
2. An Exception will be thrown if there is no certificate.
3. Because there could be more than one certificate, the checker inspects them one by one in a loop.
4. Get the next certificate in the queue and validate it. The validation process includes certificate format version checking, algorithm checking, validity period checking, signature validation, and revocation validation. If succeeds, proceed to the next step, otherwise go to step 8.
5. Locate the certificate issuer in the trust policy by matching its public key. If the certificate

issuer can be found, then go to step 7, otherwise proceed to the next step.

6. Find a position in the trust forest and the trust policy.
7. If the certificate content complies with the trust policy, then add it to the trust forest, otherwise proceed to the next step.
8. If there are more certificates in the queue to be processed, go to step 4, otherwise proceed to the next step.
9. Now all the certificates have been processed and we have a constructed trust forest. If the trust forest is empty, i.e., none of the certificates is trusted, then it fails, otherwise proceed to the next step.
10. Find rooted *leaves* in the trust forest. If there are no rooted *leaves*, then it fails, otherwise proceed to the next step.
11. If the request complies with the *leaves*, then it succeeds, otherwise it fails.

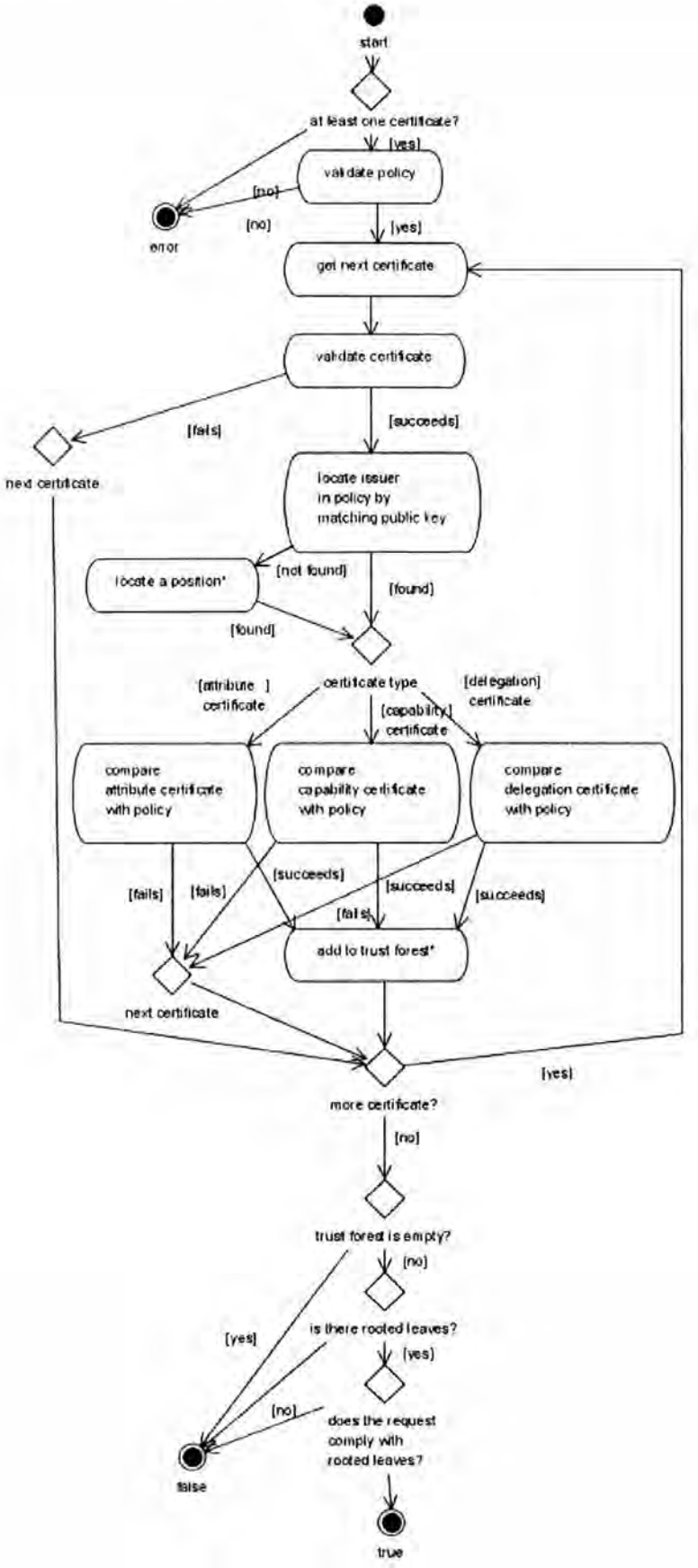

Figure 4-4 the activity diagram of the compliance checking procedure

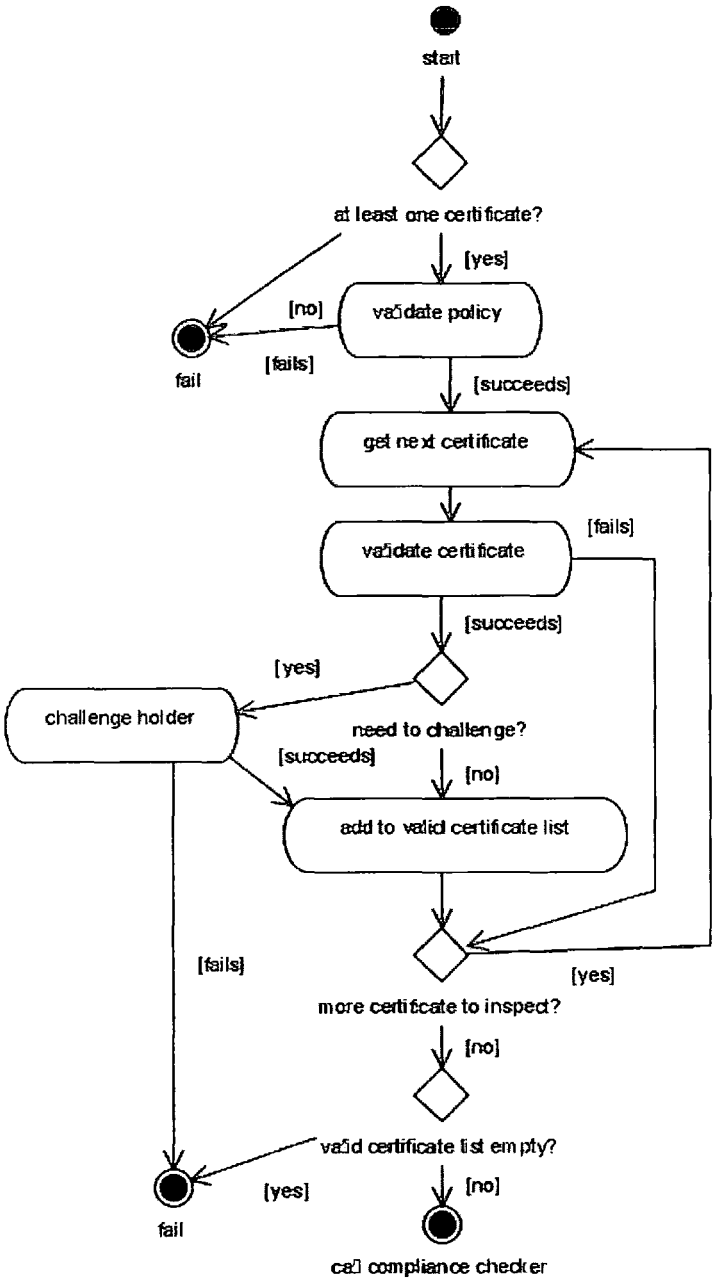

Figure 4-5 how the application invokes the compliance checker

Figure 4-5 shows how an application invokes the compliance checker

1. There must be at least one certificate otherwise an error occurs.
2. Validating the policy. This optional step could be done at the application level or compliance checker level.
3. Get next certificate in the queue.
4. Validate the certificate. This step includes XML validation, schema compliance checking

and certificate validity verification.

5. If it needs to ensure the alleged holder of the certificate, then challenge the holder with a randomly generated text. Otherwise go to step 7.
6. If the challenge succeeds, then add it the valid certificate list, otherwise the whole process fails.
7. If there are more certificates in the queue, then go to step 3, otherwise proceed to the next step.
8. If the valid certificate list is not empty, then call the compliance checker using the certificate list and policy as parameters, Otherwise it fails.

4.4 Shared libraries

The implementation has been wrapped into a web service and a .jar file. It can be integrated into any applications as an access control component.

4.4.1 Jar library

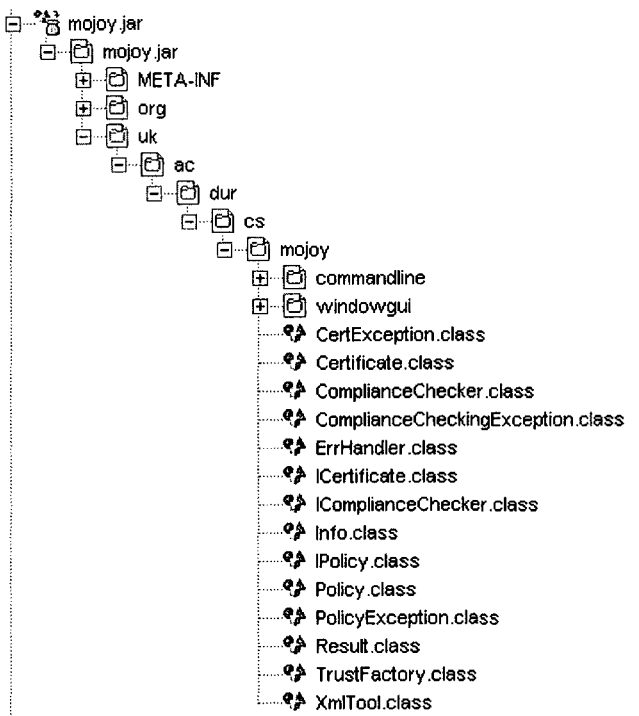

Figure 4-6 the .jar file of the implementation

Application developers can easily import the classes into the application and use the

interfaces and implementations. Figure 4-7 shows a simple example. It constructs an ICertificate object, an IPolicy object and an IComplianceChecker object. It invokes the *check* method and gets a *boolean* result.

```
import uk.ac.dur.cs.mojoy.*;


public class CommandLine {
  public static void main(String[] args){
    try{
      ICertificate cert=TrustFactory.getCertificate("certificate.xml");
      IPolicy policy=TrustFactory.getPolicy("policy.xml");
      IComplianceChecker checker=TrustFactory.getComplianceChecker();
      boolean result=checker(cert, policy);
    }catch(Exception e){
      System.out.println(e.printStackTrace());
    }
  }
}
```

Figure 4-7 an example of the usage of the compliance checker .jar library

4.4.2 Web services

The library has also been wrapped into web services using Axis, which is the successor of Apache SOAP. Our test environment of the web service is Microsoft Windows XP Professional Edition, Sun Java JDK 1.4.1. The web service server is Tomcat 4.0.6.

Deploying the web service is quite simple. Copy mojoy.war under the directory *$tomcat/webapps*, launch Tomcat, Tomcat will automatically uncompress and deploy. The URL of the deployment is *http://localhost:8080/mojoy/services/*, type the URL into the address bar of the browser and hit return, the deployment information will appear in the browser window (Figure 4-8). These messages mean that the web services had been successfully deployed and are ready to be invoked.

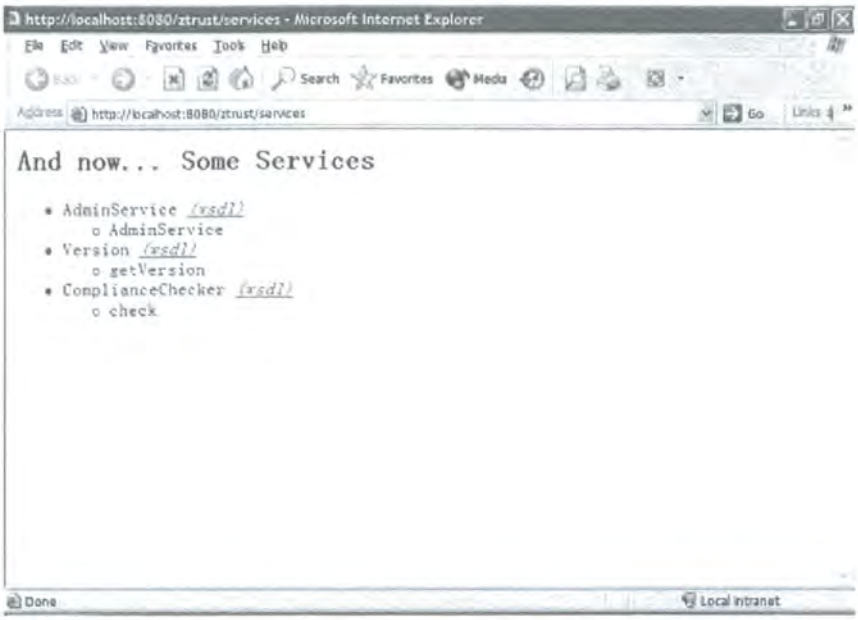

Figure 4-8 deployment of the compliance checker as web services

```
ComplianceCheckerSoapBindingStub binding = (ComplianceCheckerSoapBindingStub) new
    ComplianceCheckerServiceLocator().getComplianceChecker();
boolean value = binding.check(certificate, policy);
```

Figure 4-9 web service client side sample

Figure 4-9 shows a basic sample of the invoking of the compliance checker deployed as a web service. Compared with the .jar library, the advantages of web service are obvious:

- There is no factory class needed to manufacture the implementation.
- Server side can upgrade the web service implementation whenever they want without notifying client side.
- The client side application could be developed in any language under any platform other than Java.

4.5 Demo

The purpose of the demo is to demonstrate the work process of the compliance checker. The demo itself is a calling application, the compliance checker acts as the access control component. The application (demo) accepts outside user request and passes it, together with some supporting documents, to the access control component (compliance checker). The access control component examines the relevant documents and returns an access control decision. The application shows

the final decision to the user. The input includes user request, user certificate, local trust policy and environmental parameters. The access control decisions are *permit*, *deny* or *cannot determine*.

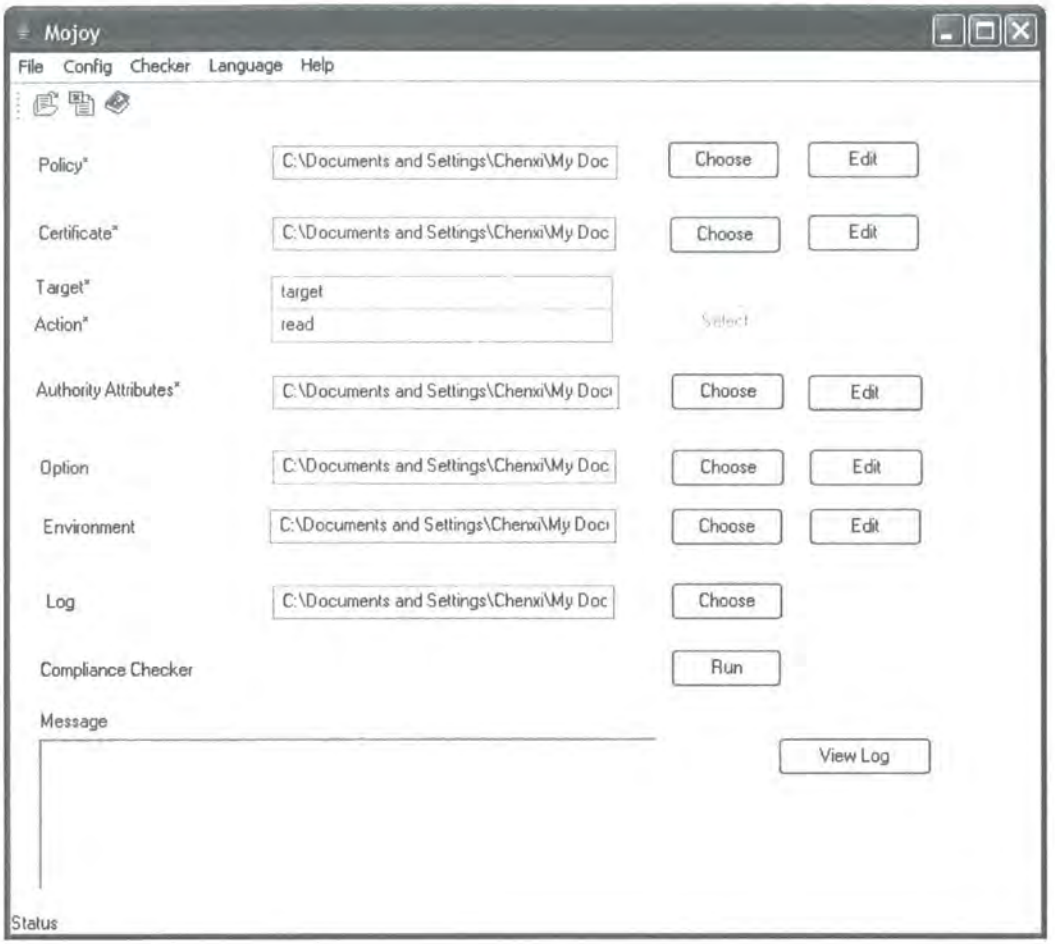

Figure 4-10 the Java demo of the compliance checker

The demo of the compliance checker is written in Java Swing. The application has been wrapped in one JAR file, *mojoywindow.jar*. The test environment is Microsoft Windows XP Professional Edition, Sun Java SDK 1.4.1, Apache Xerces XML parser. The compliance checker's version is 0.1. Double click the file *mojoywindow.jar* or type *Java -jar mojoywindow.jar* at the command line prompt will launch the application (Figure 4-10).

Please follow the instructions to invoke the checking process.

1. Choose the policy file. Policy is the locally trusted security policy. Users can type the full path and file name into the box, or use the *Choose* button to open a file chooser dialog and select a file. Currently, the compliance checker only supports single policy file, later versions will support multiple policies. The default policy is *policy.xml* under the present directory. The *Edit* button allows the users to view and edit the file content (Figure 4-11).

2. Choose the certificate file. Certificate is the user's evidence to support his request. Currently, the compliance checker only supports one certificate. It is expected that later versions will support multiple certificates. The default certificate is *cert.xml*.
3. Choose the request. The request consists of a target and an action. The target is the object that the user wants to access. The action is what the user wants to perform on the object.
4. Choose authority attribute file. The authority attribute file contains the attributes of the certificate issuers. They are retrieved from other relevant certificates and stored in the file. The process of how they are retrieved is omitted here. The default file for is *auattr.xml*.
5. Choose option file. The option file specifies the settings of the compliance checker. It specifies the logging level, whether to perform format validity check, the version of the XML parser, etc. The compliance checker will use default settings without the option file being specified. The default file is *option.txt*. This setting is optional.
6. Choose environmental setting file. The environmental setting file contains environmental settings, such as time and IP address. These environmental parameters are collected by the application and stored in the file. The default file is *env.txt*. This setting is optional.
7. Choose log file. The log file records the debug, warning and/or error messages generated during the process of compliance checking. It helps the administrator to determine the problems during compliance checking. If it is left empty, then no message will be logged. The default file is *log.txt*. This setting is optional.
8. Start compliance checker. Click the *Run* button to start the compliance checker.
9. The message box will show the process and the result of the compliance checking: *permit*, *deny* or *cannot determine*. The log file contains the comprehensive information of the error messages, for instance, which part of the certificate/policy does not comply with the schema, or what target/action is not recognized. Users can view the log file by clicking the *View Log* button.

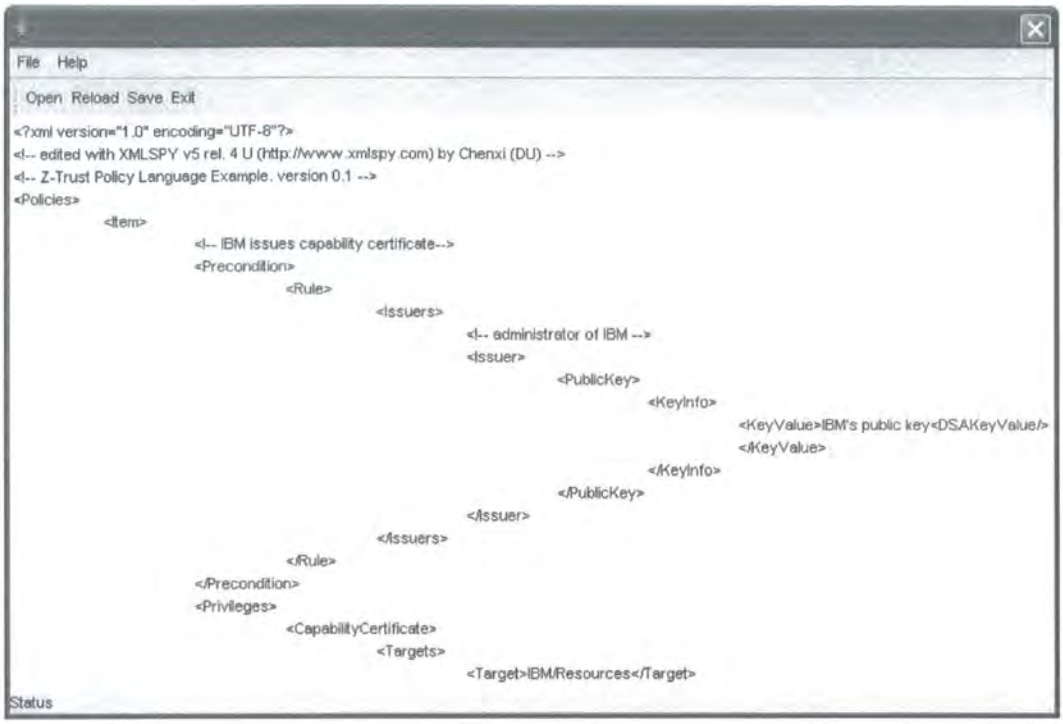

Figure 4-11 editor dialog window

The editor dialog window is opened by the demo. In the editor window, user can open, view, edit and save file.

4.6 Performance and security issues

There have always been performance concerns about PKI. Our architecture is not an exception since the certificate is protected by digital signature based on PKI. Theoretically, there are two bottlenecks during the process of compliance checking. The first one is certificate validation, including certificate digital signature verification and revocation check. Digital signature verification includes a series of XML canonical computing, message digest computing, and public key encryption. And these complex algorithms usually take up a lot of computing resources, the execution time is proportional to the size of the certificate. Because the certificate is usually very small compared to the size of policy, the execution time could be regarded as a static and small value. Certificate revocation check includes referring back to the issuing authority and checking whether it has been explicitly revoked before the validity date. The second bottleneck is the compliance checking process. Because it involves a double loop, the execution time is linearly proportional to the size of certificate and policy. As a solution to alleviate the burden of

revalidating and processing these certificates, a certificate repository could be established to store the valid certificates for a short period, for example, six hours. Within this period, certificate would not have to be revalidated and hence expedite the performance of the whole system.

XML parsing is not a problem at all. According to our test, validating a 3MB XML file only takes up 2-3 seconds. Our test environment is Pentium 4 2.8GHz, 512MB memory, 40GB hard drive, Windows XP Professional Edition, Sun Java JDK 1.4, and Apache Xerces XML Java parser.

4.7 Implementation limitations

Our implementation has several limitations due to limited time. For instance, the function of certificate validation is not included. The XML parser is able to check XML validity, the certificate and policy tested in our experiments are assumed to be valid by itself. Actually there are several off the peg software packages available to carry out the task. For instance, Apache XML security library provides both Java and C++ implementations that comply with the W3C standards [21][22]. Also, the function of challenging the holder is not implemented. Implementing the function of challenge should be careful of replay attack, in which the attacker intercepts the response sent back by the authentic holder and resends it to the server to impersonate the holder. To prevent such attacks, server should append a nonce to the challenging message. Replies with an obsolete nonce should be discarded. Implementing the construction of trust forest from loads of certificates and finding out a trust train from the constructed trust forest is huge work. It involves privilege delegation and empowerment, interpretation of various certificates and enforcement of local trust policy. Our implemented is able to deal with one certificate and one policy. The algorithm of dealing with two or more certificates is very complicated and is left to future work.

4.8 Summary

The task of the compliance checker is to take in the user request, user's certificate, the policy and environmental parameters and produce an access control decision. We have developed a Java implementation of the compliance checker in this chapter. A demo based on Java Swing is also developed to demonstrate how the compliance checker works. The checker is designed to be flexible to be incorporated into any applications.

5 Application Case Study

5.1 Resource Sharing Union

Durham.org and newcastle.org are two organizations. They provide resources for their registered users. Both organizations have their own security regulations to protect their online resources. Only registered users are allowed to access the resources. The resources are divided into three categories: public, private and premium. Public resources are available to everybody (including non-registered users). These resources include weather forecast, Internet search and web storage, etc. Private resources are for registered users only. Premium resources are restricted to premium users. Premium resources will charge a fee according to usage. These resources include financial information real time report, advanced Internet search, tourist discount, etc.

The present situation is that every registered user has a unique account and a password to log onto the system. To protect their resources, the two organizations have adopted different Role Based Access Control (RBAC) mechanisms. In durham.org, four kinds of roles have been defined. They are *guests*, *members*, *premium members* and *administrators* (Table 5-1). *Guests* are those non-registered users whose actions are restricted to "read public information". *Members* are those registered users who have access to the free resources. *Premium members* are upgraded from *members*. They subscribe value-added services and pay an annual fee to enjoy them. *Administrators* perform supervision tasks. They set up local security policies, issue certificates to registered members, etc. In newcastle.org, five kinds of roles have been defined. They are *guests*, *users*, *power users*, *staffs* and *managers* (Table 5-2). *Guests* are those non-registered users whose actions are restricted to browse public information. *Users* are those registered users who have access to the free resources. *Power users* are upgraded from *users*; they subscribe value-added services and pay an annual fee to enjoy them. *Staffs* have a special privilege to access "staff-only" information. *Managers* perform system routine tasks. They set up local security policies, issue certificates to registered members, etc.

Table 5-1 durham.org's RBAC system

ROLES	PERMISSIONS
Guests	Browse public information
Members	Access to internal information
Premium members	Access to pay services
Administrators	System supervision

Table 5-2 newcastle.org's RBAC system

ROLES	PERMISSIONS
Guests	Browse public information
Users	Access to internal information
Power users	Access to advanced information
Staff	Access to staff-only information
Managers	System supervision

Both durham.org and newcastle.org have advantages and disadvantages of resources they offer. For example, durham.org provides tourist discount information but newcastle.org does not, while newcastle.org provides latest financial real time report but durham.org does not. Developing these services independently is time consuming and financially unacceptable. The best solution would be sharing resources with each other. The members of durham.org will be able to access the online resources of the newcastle.org and vice versa.

The requirements of the new architecture are:

- The original security architecture of the organizations should be preserved so that those previous registered users' privileges remain intact.
- The users of either side could access peer's resources without an additional registration.
- The new architecture should have the ability to incorporate more potential organizations into the system without change to the existing architecture. The joining/leaving of the individual organization does not affect the whole system.
- All the resources of the two organizations should be uniformly presented to the users as if they are in the same location.
- The organizations are independent of each other; each maintains its own policy.

5.2 Incompetence of the previous access control models

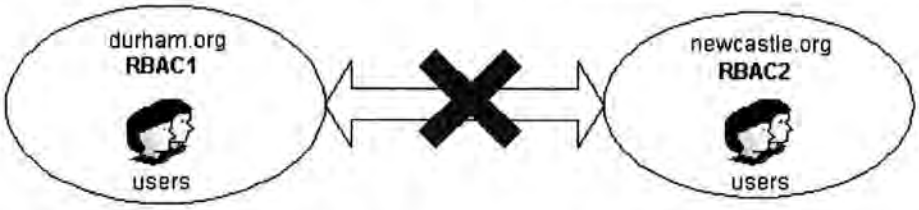

Figure 5-1 incompatible RBAC prohibits the unification

RBAC, the current access control model of both systems, does not satisfy the new requirements of the upgrade. The drawback is that the account is only recognized locally and is not effective outside of the scope of the organization. At the first glance, those two online organizations resemble each other a bit, but a further careful study reveals their inner incompatibilities. For example, Alice is a registered user of durham.org, her account is *alice*. She is a *premium member*, i.e. the account *alice* has been assigned the role *premium members*, or *alice<-premium members*. Everything is fine if she stays within durham.org, but when she goes to newcastle.org, the account *alice* is not recognized, thus all her request will be rejected. Merging all the accounts of the two organizations is not acceptable because it takes too much time and effort and there could be account name conflict. Appending domain name to the tail could be a solution, for instance *alice@durham*, but incompatible RBAC systems thwart it. The access control system of newcastle.org will find that the role *premium members* assigned to *alice@durham* is not recognized, therefore all her requests will be denied. Even if the names of the roles are identical, for instance, both organizations have defined the role *guests*, the privileges assigned to the roles could be radically different. Furthermore, unboundedly importing accounts and roles into the current system from newly joined organizations could be an undue burden. Therefore we can draw the conclusion that simply merging the two or more RBACs is completely impractical under the distributed computing environment. We must find a new solution.

5.3 Our solution

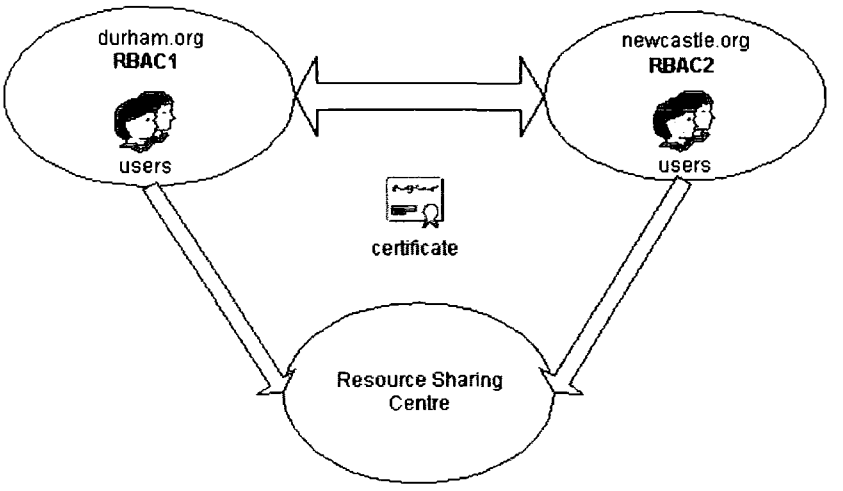

Figure 5-2 our solution for the Resource Sharing Union

A Resource Sharing Centre (RSC) will be set up to help the communication between the organizations (Figure 5-2). The organizations publish resources to the RSC. Users look for published information on the RSC. Five basic actions can be performed on the RSC. They are *list*, *search*, *info*, *subscribe* and *get*. *List* returns all the available resources. *Search* returns resources by keywords. *Info* returns the description of the resource. *Get* redirects the user to the real location of the resource.

RSC allows requests from registered users of the allied organizations. Users must produce an attribute certificate to prove that he/she is a member of the allied organization. RSC validates the certificate. Validation process divides into the following steps. First of all, the system checks the certificate format is recognized and strictly follows one of the standards. The second step ensures that the certificate is within its validity period by comparing the *valid after time*, *valid before time* and *current time* of the system. The third step computes the canonicalized value of the certificate, the message digest value, the signature with the public key contained in the certificate and compares it with the original signature. The fourth step checks whether the certificate has been revoked. If all of the four steps pass, then the certificate is regarded as valid. Then the server challenges the certificate holder to verify that he is the owner of the corresponding private key, usually by sending a message encrypted by the public key contained in the certificate, the message is comprised of a random sequence plus nonce. If the alleged holder could successfully decrypt and send back the original information, then the server is convinced that the holder is authentic.

Next the RSC examines whether the certificate issuer is trusted. It scans its local trust policy to find a *root of trust* (see chapter 2.1), usually by matching public key. If the certificate is within legitimate scope, then the certificate is trusted. If a direct trust relationship cannot be established, the server will try to find a chain of certificates that could be finally traced back to a *root of trust*. The certificate is trusted either through a direct trust or a trust chain.

The user begins to find resources on the RSC after trust has been established. The RSC is only a directory which lists all the resources and will redirect the user to the real location upon request. RSC acts as a trusted third party between the users and the organizations. For instance, it could certify the credibility of the users by double signing the user's certificate, or issues certificate directly to the user, or issues certificate to the user's belonged organization. The former two create a direct trust while the last one produces a trust chain.

Upon receiving request from the user, the service provider first validates the certificate. Then maps the user to the local RBAC system according to SLA, or makes access decision directly from the user's capability certificate. Finally, the user accesses the resources.

The advantages of our new architecture are:

- The allied organizations do not have to change the original access control model, they can preserve the old security structure, thus to the most extent avoid the disorder caused by the introduction of the new system.
- Members do not have to register with every organization. They get certificate from the organization they belonged to. Their authenticity is guaranteed by the certificate issuer as long as the issuer is trusted by the other organizations.
- The organizations are independent of each other. Each one has its own trust policy that defines the credibility and trust level of the other organizations.
- Organizations do not have to know each other in advance. The precondition is that all allied organizations must at least trust one RSC. RSC guarantees the credibility of all the allied organizations by issuing certificate to them.
- Newly joined organizations could easily be incorporated into the circle. All they have to do is getting a certificate from the RSC. RSC issues delegation certificate to the new organization. All the allied members trust the RSC and thus accept the newly joined organization.

5.4 Assumptions

It is necessary for us to make some assumptions of the designed architecture.

- The RSC is trusted by all the allied organizations in the architecture. All the certificates issued by RSC are trusted by the organizations.
- The syntax and semantics of the certificates and the policy are understood by all the allied organizations and RSC. There would be no misunderstandings and/or misinterpretations.
- The certificate and policy are written in well-formed XML. They are semantically healthy and will not cause confusions.
- All the registered users of the organizations have the required certificates to identify themselves.
- Organizations and RSC are willing to issue certificates to the users upon request. It is also the individual organization's responsibility to protect their users' privacy.
- Users can store their private keys securely and prove the ownership upon challenge.
- The server protects its local trust policy from being tampered.
- The server will automatically collect missing certificates.
- The privacy protection of the certificate and policy will not be discussed. We assume that all the certificates in the application case study are disclosed on a basis of willingness and do not involve any privacy issues.

6 Scenarios and Experiments

6.1 Scenario 1 (certificate and RBAC)

Alice is a registered member of durham.org. She has an account *alice* and has been assigned the role *member*. She can search for information in durham.org. Now she wants to find information about *computer virus* in newcastle.org. First of all, she needs to get a certificate from durham.org that certifies her present situation in durham.org (Figure 6-1). The certificate says that Alice is a registered member of durham.org. Her position is *member*. The validity period of the certificate is from 01/01/2004 to 01/01/2010. Alice's authenticity is proved by her public key.

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- Alice's attribute certificate issued by durham.org -->
<Certificate xmlns="http://www.dur.ac.uk/chenxi.huang/mojoy0.1">
    <Attributes>
        <Attribute>
            <Name>organization</Name>
            <Value>durham.org</Value>
        </Attribute>
        <Attribute>
            <Name>position</Name>
            <Value>member</Value>
        </Attribute>
    </Attributes>
    <Conditions>
        <Condition>
            <Constraints>
              <Constraint>
                <TimeConstraint>
                    <StartTime>00:00:00 01/01/2004</StartTime>
```

```
                    <EndTime>00:00:00 01/01/2010</EndTime>
                </TimeConstraint>
            </Constraint>
        </Constraints>
    </Condition>
</Conditions>
<Holders>
    <Subject>
        <PublicKey>Alice's public key</PublicKey>
    </Subject>
</Holders>
<Issuers>
    <Subject>
        <PublicKey>durham.org's public key</PublicKey>
    </Subject>
</Issuers>
<Signature>
    signature of the certificate
</Signature>
</Certificate>
```

Figure 6-1 Alice's attribute certificate issued by durham.org

For the second step, she submits her certificate to the Resource Sharing Centre. RSC validates the certificate; it verifies the certificate's well-formedness, signature, validity and the authenticity of the holder. RSC recognizes durham.org because it is an allied organization of the Resource Sharing Union. Then RSC looks up its local trust policy (Figure 6-2). The policy says that durham.org is a trusted organization; the attribute certificate issued by durham.org is trusted. RSC locates the entry of durham.org by matching the public key and finds out Alice's attribute certificate is within the legitimate scope. Then RSC grants Alice the privileges according the local attribute-permission mapping policy. In this case, Alice will be granted the ability to search for

information within the Union.

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- RSC's local security policy -->
<Policy xmlns="http://www.dur.ac.uk/chenxi.huang/mojoy0.1">
    <Rules>
        <Rule>
            <!--durham.org & newcastle.org can issue attribute & capability certificate-->
            <Conditions>
                <Condition>
                    <Issuers>
                        <Subject>
                            <PublicKey>
                                durham.org's public key
                            </PublicKey>
                        </Subject>
                        <Subject>
                            <PublicKey>
                                newcastle.org's public key
                            </PublicKey>
                        </Subject>
                    </Issuers>
                </Condition>
            </Conditions>
            <Privileges>
                <Privilege>
                    <Capabilities>
                        <Capability>
                            <Targets>
                                <Target>rsc/searchservice</Target>
                            </Targets>
```

```
                    <Actions>
                        <Action>read</Action>
                        <Action>execute</Action>
                    </Actions>
                </Capability>
            </Capabilities>
            <Attributes>
                <AnyAttribute/>
            </Attributes>
        </Privilege>
    </Privileges>
</Rule>
</Rules>
</Policy>
```

Figure 6-2 RSC's local trust policy

For the third step, Alice enters the keyword *computer virus* into the search engine and gets a list of results. She is interested in two results, one is an ordinary news report and the other is a pay service, both of which are located on a remote server of newcastle.org. RSC redirects her to newcastle.org. When Alice arrives at newcastle.org, she produces her attribute certificate (Figure 6-1) to the access control system of newcastle.org. The access control system validates the certificate and then looks up its local trust policy (Figure 6-3). The policy says that durham.org is a trusted organization; the attribute certificate issued by durham.org is trusted. The access control system locates the entry of durham.org by matching the public key. It finds out that Alice's attribute certificate issued by durham.org is trusted.

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- newcastle.org's local security policy -->
<Policy xmlns="http://www.dur.ac.uk/chenxi.huang/mojoy0.1">
    <Rules>
        <Rule>
```

```
<!-- durham.org issues attribute certificate -->
<Conditions>
    <Condition>
        <Issuers>
            <Subject>
                <PublicKey>
                        durham.org's public key
                </PublicKey>
            </Subject>
        </Issuers>
    </Condition>
</Conditions>
<Privileges>
    <Attributes>
        <Attribute>
            <AnyAttribute/>
        </Attribute>
    </Attributes>
    <Capabilities>
        <Capability>
            <Targets>
                <Target>newcastle.org/private/a</Target>
                <Target>newcastle.org/private/b</Target>
            </Targets>
            <Actions>
                <Action>read</Action>
                <Action>execute</Action>
            </Actions>
        </Capability>
    </Capabilities>
```

```
            </Privileges>
        </Rule>
    </Rules>
</Policy>
```

Figure 6-3 newcastle.org's local trust policy

Then Alice's position will be mapped to the local RBAC system according to SLA (the design of SLA is beyond the topic of this thesis). In this case, Alice will be assigned the role *users* in the local system. Then access control will be handed over to local RBAC system to carry on. The role of *users* has been assigned the permissions to access the *computer virus* information, Alice will be granted access right to the information.

RBAC by itself does not provide fine-grained access control. The server needs extra information of the requester in order to finely control the resources. For instance, the resources are pay services. These services are restricted to the premium members of newcastle.org and they must pay to enjoy the services. Alice is also required to pay before she could use the service. She could pay either at durham.org or RSC, and in return she gets a certificate (Figure 6-4). The certificate says that Alice is the holder. She is able to read the resources identified by the URI *newcastle.org/private/a*. Durham.org is the issuer. The validity period of the certificate is from 01/01/2004 to 01/01/2005.

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- Alice's capability certificate -->
<Certificate xmlns="http://www.dur.ac.uk/chenxi.huang/mojoy0.1">
    <Capabilities>
        <Capability>
            <Targets>
                <Target>newcastle.org/private/a</Target>
            </Targets>
            <Actions>
                <Action>read</Action>
            </Actions>
```

```
        </Capability>
    </Capabilities>
    <Conditions>
        <Condition>
            <Constraints>
                <Constraint>
                    <TimeConstraint>
                        <StartTime>00:00:00 01/01/2004</StartTime>
                        <EndTime>00:00:00 01/01/2005</EndTime>
                    </TimeConstraint>
                </Constraint>
            </Constraints>
        </Condition>
    </Conditions>
    <Holders>
        <Subject>
            <PublicKey>
                    Alice's public key
            </PublicKey>
        </Subject>
    </Holders>
    <Issuers>
        <Subject>
            <PublicKey>
                    durham.org's public key
            </PublicKey>
        </Subject>
    </Issuers>
    <Signature>
        signature of the certificate
```

```
    </Signature>
</Certificate>
```

Figure 6-4 Alice's capability certificate that confines her to the specific service

Alice submits it to newcatle.org. The access control system validates the certificate and looks into its local trust policy (Figure 6-3). The policy says that durham.org is a trusted authority, it is able to issue certificates to its users but the targets should be confined to *newcastle.org/private/a* and *newcastle.org/private/b*, the actions should be confined to *read* and *execute*. The access control system verifies that the public key of the certificate issuer is durham.org and the capabilities contained in the certificate are within the confined scope. Alice's request is to *read* the resource *newcastle.org/private/a*. The request is allowed by the certificate and the policy, therefore the access will be allowed. She finally gains access to the *computer virus* information.

6.2 Scenario 2 (environmental factors)

Sometimes the service providers require placing environmental constraints. These constraints are independent of the users and cannot be represented through RBAC. Environmental constraints include time and location constraints. An example is that newcastle.org wants to restrict the access from the users of durham.org within the period from Monday to Friday, 9 am to 2 pm, and the request must be initiated from a recognized IP address. The policy is defined in Figure 6-5. The policy says that durham.org is a trusted issuer and it is able to issue certificates. The valid capabilities include *read* permission to *newcastle.org/public* and *read, execute* permissions to *newcastle.org/premium*. Any attributes are permitted. The certificates are restricted by the constraints. The constraints include date, time and IP address. The permitted time period is from 9 am to 5 pm, specified by the local time of newcastle.org. The request must be initiated from the IP address segment 129.234.155.0/24, which denotes the IP address segment is 129.234.155.0 and the subnet mask is 255.255.255.0.

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- Newcastle.org's local trust policy with environmental constraints -->
<Policy xmlns="http://www.dur.ac.uk/chenxi.huang/mojoy0.1">
    <Rules>
```

```
<Rule>
    <Conditions>
        <Issuers>
            <Subject>
                <PublicKey>
                        durham.org's public key
                </PublicKey>
            </Subject>
        </Issuers>
        <Constraints>
          <Constraint>
            <IPConstraint>
                129.234.155.0/24
            </IPConstraint>
            <TimeConstraint>
                <StartTime>09:00:00</StartTime>
                <EndTime>17:00:00</EndTime>
            </TimeConstraint>
          </Constraint>
        </Constraints>
    </Conditions>
    <Privileges>
        <Capabilities>
            <Capability>
                <Targets>
                    <Target>newcastle.org/public</Target>
                </Targets>
                <Actions>
                    <Action>read</Action>
                </Actions>
```

```
                    </Capability>
                    <Capability>
                        <Targets>
                            <Target>newcastle.org/premium</Target>
                        </Targets>
                        <Actions>
                            <Action>read</Action>
                            <Action>execute</Action>
                        </Actions>
                    </Capability>
                </Capabilities>
                <Attributes>
                    <AnyAttribute/>
                </Attributes>
            </Privileges>
        </Rule>
    </Rules>
</Policy>
```

Figure 6-5 newcastle.org's local trust policy with environmental constraints

These constraints could also be expressed in the certificate by the issuer (Figure 6-6). The certificate is issued by durham.org, it says that Alice is able to *read* the resource *newcastle.org/public*, but her access is restricted within the period from 9 am to 5 pm, and the access must be initiated from a computer with a recognized IP address within the segment 129.234.155.0/24.

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- Alice's certificate with constraints -->
<Certificate xmlns="http://www.dur.ac.uk/chenxi.huang/mojoy0.1">
    <Capabilities>
        <Capability>
```

```
            <Targets>
                <Target>newcastle.org/public</Target>
            </Targets>
            <Actions>
                <Action>read</Action>
            </Actions>
        </Capability>
    </Capabilities>
    <Conditions>
        <Condition>
            <Constraints>
                <Constraint>
                    <TimeConstraint>
                        <StartTime>09:00:00</StartTime>
                        <EndTime>17:00:00</EndTime>
                    </TimeConstraint>
                    <IPConstraint>
                        129.234.155.0/24
                    </IPConstraint>
                </Constraint>
            </Constraints>
        </Condition>
    </Conditions>
    <Holders>
        <Subject>
            <PublicKey>
                Alice's public key
            </PublicKey>
        </Subject>
    </Holders>
```

```
<Issuers>
    <Subject>
        <PublicKey>
            durham.org's public key
        </PublicKey>
    </Subject>
</Issuers>
<Signature>
    signature of the certificate
</Signature>
</Certificate>
```

Figure 6-6 Alice's certificate that contains environmental constraints

6.3 Scenario 3 (delegation and empowerment)

Another organization leeds.org wants to join the Resource Sharing Union, so that all the three organizations could share their resources together. Leeds.org has its own RBAC access control system that is incompatible with the others'. Four types of roles have been defined: *visitors*, *students*, *staff* and *root*. *Visitors* are those outsiders who do not have registered accounts with leeds.org; they can only browse public information. *Students* are those who have registered and have been assigned an account and a password to log onto the system; they can access to both public and internal information. *Staff* users have been assigned a special privilege to the *staff only* section. *Root* users are the system security administrators; they perform system maintenance tasks, such as making policies, manage user accounts, etc.

Table 6-1 the RBAC system of leeds.org

ROLES	PERSMISSIONS
Students	Access to public and private information
Visitors	Browse public information
Staff	Access to information that is only available to staff

Root	Manages the security issues

Bob is a registered member of leeds.org. He has an attribute certificate from leeds.org (Figure 6-7). If he goes to newcastle.org and wants to access the resources, he will be rejected because the issuer of the certificate, leeds.org, is a stranger to newcastle.org. The certificate issued by leeds.org will not be trusted by newcastle.org.

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- Bob's attribute certificate issued by leeds.org -->
<Certificate xmlns="http://www.dur.ac.uk/chenxi.huang/mojoy0.1">
    <Attributes>
        <Attribute>
            <Name>organization</Name>
            <Value>leeds.org</Value>
        </Attribute>
        <Attribute>
            <Name>position</Name>
            <Value>member</Value>
        </Attribute>
    </Attributes>
    <Conditions>
        <Condition>
            <Constraints>
              <Constraint>
                <TimeConstraint>
                    <StartTime>01/01/2004 00:00:00</StartTime>
                    <EndTime>01/01/2010 00:00:00</EndTime>
                </TimeConstraint>
              </Constraint>
            </Constraints>
        </Condition>
```

```
    </Conditions>
    <Holders>
        <Subject>
            <PublicKey>
                Bob's public key
            </PublicKey>
        </Subject>
    </Holders>
    <Issuers>
        <Subject>
            <PublicKey>
                leeds.org's public key
            </PublicKey>
        </Subject>
    </Issuers>
    <Signature>
        signature of the certificate
    </Signature>
</Certificate>
```

Figure 6-7 Bob's attribute certificate issued by leeds.org

The easiest way to incorporate leeds.org into the union is to get a delegation certificate (Figure 6-8) from the RSC because RSC is trusted by all the allied organizations. The certificate says that leeds.org is able to issue any attribute and capability to its members. The validity period is from 01/01/2004 to 01/01/2010.

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- leeds' delegation certificate issued by RSC -->
<Certificate xmlns="http://www.dur.ac.uk/chenxi.huang/mojoy0.1">
    <Privileges>
        <Privilege>
```

```
            <Attributes>
                <AnyAttribute/>
            </Attributes>
            <Capabilities>
                <AnyCapability/>
            </Capabilities>
        </Privilege>
    </Privileges>
    <Conditions>
        <Condition>
          <Constraints>
            <Constraint>
                <TimeConstraint>
                    <StartTime>00:00:00 01/01/2004</StartTime>
                    <EndTime>00:00:00 01/01/2010</EndTime>
                </TimeConstraint>
            </Constraint>
          </Constraints>
        </Condition>
    </Conditions>
    <Holders>
        <Subject>
            <PublicKey>
                leeds' public key
            </PublicKey>
        </Subject>
    </Holders>
    <Issuers>
        <Subject>
            <PublicKey>
```

```
                RSC's public key
            </PublicKey>
        </Subject>
    </Issuers>
    <Signature>
        signature of the certificate
    </Signature>
</Certificate>
```

Figure 6-8 leeds.org's delegation certificate issued by RSC

Newcastle.org's trust policy (Figure 6-9) says that RSC is a trusted entity, it is allowed to issue certificate to the allied organizations. The privileges that can be delegated include any attribute and some specific capabilities. Bob goes to newcastle.org and requests for the resources. He produces his attribute certificate (Figure 6-7) and leeds.org's delegation certificate issued by RSC (Figure 6-8). The access control system of newcastle.org looks up its local trust policy, finds out that the RSC is a trusted entity and the delegation certificate issued by the RSC is trusted. Leeds.org is allowed to issue attribute certificate. Therefore a trust chain could be found, which is RSC -> leeds.org -> Bob. Newcastle.org establishes trust with leeds.org through RSC and as a result Bob's certificate is accepted. Bob will be mapped to a local role according to a series of mapping rules. In this case he will be granted the role *users* (trust is usually established at the minimal degree). The rest of the access control procedure will be handed over to local RBAC system. Finally Bob gets access to the resource.

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- newcastle.org's trust policy that allows empowerment -->
<Policy xmlns="http;//www.dur.ac.uk/chenxi.huang/mojoy0.1">
    <Rules>
        <Rule>
            <Conditions>
                <Condition>
                    <Issuers>
```

```
                    <Subject>
                        <PublicKey>
                            RSC's public key
                        </PublicKey>
                    </Subject>
                </Issuers>
            </Condition>
        </Conditions>
        <Privilege>
            <Attributes>
                <Attribute>
                    <Name>Resource Sharing Union</Name>
                    <Value>member</Value>
                </Attribute>
            </Attributes>
        </Privilege>
    </Rule>
    <Rule>
        <Conditions>
            <Condition>
                <Issuers>
                    <Subject>
                        <Attributes>
                            <Attribute>
                                <Name>Resource Sharing Union</Name>
                                <Value>member</Value>
                            </Attribute>
                        </Attributes>
                    </Subject>
                </Issuers>
```

```
            </Condition>
        </Conditions>
        <Privilege>
            <Attributes>
                <AnyAttribute/>
            </Attributes>
            <Capabilities>
                <Capability>
                    <Targets>
                        <Target>newcastle.org/public</Target>
                    </Targets>
                    <Actions>
                        <Action>read</Action>
                    </Actions>
                </Capability>
            </Capabilities>
        </Privilege>
    </Rule>
</Rules>
</Policy>
```

Figure 6-9 newcastle.org's trust policy that allows empowerment

Unconstrained delegation of privilege is often dangerous because of the jeopardy of being compromised. Hence it should be carefully dealt with. In our model of empowerment, individual entity could finely control the scope of delegation by specifying the privileges assigned to the delegatee in its local trust policy. For instance, newcastle.org's trust policy (Figure 6-9) says that RSC is a trustworthy delegation authority, and it is permitted to delegate *read* permission to *newcastle.org/public* to any subject.

Bob's capability certificate (Figure 6-10) issued by leeds.org grants him *read* permission to the private information (*newcastle.org/private*) and public information (*newcastle.org/public*) of

newcastle.org. A trust chain could be found to support him, which is RSC -> leeds.org -> Bob. According to the trust policy, the former is out of the valid scope and is not permitted, but the latter is allowed. Therefore, Bob's request to read *newcastle.org/private* will be denied but his request to read *newcatle.org/public* will be allowed.

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- Bob's capability certificate issued by leeds.org -->
<Certificate xmlns="http://www.dur.ac.uk/chenxi.huang/mojoy0.1">
    <Capabilities>
        <Capability>
            <Targets>
                <Target>newcastle.org/private</Target>
            </Targets>
            <Actions>
                <Action>read</Action>
            </Actions>
        </Capability>
        <Capability>
            <Targets>
                <Target>newcastle.org/public</Target>
            </Targets>
            <Actions>
                <Action>read</Action>
            </Actions>
        </Capability>
    </Capabilities>
    <Conditions>
        <Condition>
            <Constraints>
              <Constraint>
                <TimeConstraint>
```

```
                    <StartTime>00:00:00 01/01/2004</StartTime>
                    <EndTime>00:00:00 01/01/2010</EndTime>
                </TimeConstraint>
            </Constraint>
        </Constraints>
    </Condition>
</Conditions>
<Holders>
    <Subject>
        <PublicKey>
                Bob's public key
        </PublicKey>
    </Subject>
</Holders>
<Issuers>
    <Subject>
        <PublicKey>
                leeds.org's public key
        </PublicKey>
    </Subject>
</Issuers>
<Signature>
    signature of the certificate
</Signature>
</Certificate>
```

Figure 6-10 Bob's capability certificate issued by leeds.org

6.4 Scenario 4 (trust empowerment)

Leeds.org could be incorporated into the union with the concept we termed as *trust*

empowerment. Leeds.org gets a membership attribute certificate from RSC (Figure 6-11), it says that leeds.org is an allied organization of the Resource Sharing Union.

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- leeds.org's attribute certificate issued by RSC -->
<Certificate xmlns="http://www.dur.ac.uk/chenxi.huang/mojoy0.1">
    <Attributes>
        <Attribute>
            <Name>Resource Sharing Union</Name>
            <Value>member</Value>
        </Attribute>
    </Attributes>
    <Conditions>
        <Condition>
            <Constraints>
              <Constraint>
                <TimeConstraint>
                    <StartTime>00:00:00 01/01/2004</StartTime>
                    <EndTime>00:00:00 01/01/2010</EndTime>
                </TimeConstraint>
              </Constraint>
            </Constraints>
        </Condition>
    </Conditions>
    <Holders>
        <Subject>
            <PublicKey>
                    leeds' public key
            </PublicKey>
        </Subject>
    </Holders>
```

```
<Issuers>
    <Subject>
        <PublicKey>
                RSC's public key
        </PublicKey>
    </Subject>
</Issuers>
<Signature>
    signature of the certificate
</Signature>
</Certificate>
```

Figure 6-11 leeds.org's attribute certificate issued by the RSC

Newcastle.org's local trust policy (Figure 6-9) says that RSC is trusted to entitle membership to new organizations and the members of the Resource Sharing Union is able to issue certificate to their subordinate members, thereby enabling them to visit neighbours' resources in the union. Bob goes to newcastle.org and submits his attribute certificate (Figure 6-7). The access control system checks his certificate, finds out he is from an unknown organization. Then it gets the membership certificate of leeds.org (Figure 6-11). The issuer of the membership certificate is the trusted RSC. Therefore, the trust chain is found, which is newcastle.org -> RSC -> leeds.org -> Bob. Bob will become trusted.

6.5 Analysis

We have studied four scenarios in this chapter. The first scenario discusses the possibility of preserving and utilising RBAC in a distributed system. RBAC is an effective access control model for centrally managed organizations. It successfully converts abstract responsibilities to concrete roles, making security management straightforward and easy. However, in a distributed environment, different organizations adopt different RBACs, making it difficult for a user to transfer from one security domain to another. Certificate based access control sets up a bridge to supply the gap between those different RBACs. It supplements rather than replaces RBAC hence

RBAC could be preserved in the distributed system. The two organizations, durham.org and newcastle.org, cannot incorporate each other directly because of incompatible RBAC systems. Certificate based approach supplies the gap. Roles are stored in certificates as attributes. The role definitions are interpreted according to SLA. This scenario proves that certificate based approach is a very good complement of RBAC. Furthermore, capability certificate lets the certificate issuer finely control the privileges assigned to each individual.

The second scenario discusses various constraints. These constraints include time constraints and location constraints. Constraints are independent of subjects and are of use when limiting the delegated access privileges. Our approach provides a method to finely control the constraints placed on the subjects and the delegated subjects.

The third scenario discusses the scalability problem. Incorporating a new entity into an existing group has always been a challenge. In this scenario, leeds.org is a stranger to durham.org and newcastle.org. The certificates issued by leeds.org are not trusted by durham.org and newcastle.org. RSC acts as a trusted third party in the process. RSC issues delegation certificate to leeds.org, this certificate indirectly proves the credibility of the certificate issued by leeds.org. On the other hand, durham.org and leeds.org keep their own trust policies which administer the credibility of the RSC and newly joined members.

The fourth scenario discusses *trust empowerment* model. Newcastle.org trusts all the members of the Resource Sharing Union, but the identities of all the members cannot be known in advance. Therefore it trusts based on the attributes of the entities. The membership attributes are certified by the RSC. Newcastle.org further trusts the users who are certified by the members of the Resource Sharing Union. We term the process as *trust empowerment*. The process of trust empowerment can be totally and finely managed by the server.

The access control system uses our compliance checker which is implemented in Java. The compliance checker takes in the access request, the users' certificates and the server's policy. It applies a double-loop to find out whether the access request is supported by the certificate and allowed by the policy. A Boolean result will be returned to allow or deny the access request.

The achievements of our research could be applied to the healthcare domain. According to *the Medical Records Confidentiality Act of 1995* [15], the privacy of personally identifiable health information should be protected from unauthorised access.

- Health information trustees (defined as health care providers, health care plans, etc) are required to allow individual's access to any health information pertaining to the individual and to give the opportunity to correct such information. The trustees are also required to develop safeguards to protect the confidentiality of the personally identifiable health information they maintain.
- Trustees are required to obtain an individual's authorisation to disclose personally identifiable health information for purposes of payment or treatment. In addition, this title allows personally identifiable health information to be disclosed to an individual's next of kin, and the individual's name or general health status to be disclosed for directory information, as long as the individual has not objected to such disclosure.
- Personally identifiable health information may be disclosed without the individual's consent to the following authorities for legal reasons: emergency circumstances, oversight, public health, health research, judicial and administrative purposes, non-law enforcement subpoenas, law enforcement, and certified health information services.

The enforcement of the Act involves many geographically dispersed entities, including healthcare providers, patients, next of kin of patients, emergency department, the court and health information services, etc. They employ different RBACs as access control system. These entities could be regarded as the organizations in our case study and the medical records could be regarded as resources.

- Health information trustees check attributes of the patients. With the attributes contained in the certificate, trustees could easily locate each patient's personal medical record and grant the owner of the record corresponding permissions.
- Patient issues certificate to a third party, for either payment or treatment reason. With the certificate, the third party can access the patient's medical records.
- Trustees need to know the credibility of the various issuing bodies, for example, the emergency department of a hospital, a certified institutional review board (IRB) or a district court. But the identities of the trustees are often difficult to know in advance. With *trust empowerment*, the problem can be easily solved. The trustees are distinguished and trusted by their attributes which are certified by the NHS.

7 Conclusion and future work

7.1 Conclusion

In this thesis, we first reviewed traditional access control models. The most commonly used access control models were access matrix and RBAC. Access matrix uses a two dimensional matrix to represent the relationships between subjects and objects. RBAC uses role in between users and permissions to gain more flexibility. They were effective in traditional centrally managed organizations where the identities and privileges of the subjects are known and managed in one point. However, in distributed environment, different systems adopt different access control models; people frequently move from one place to another; the identities and privileges of the users are administered in dispersed locations. Traditional access control models are incompetent to deal with the new circumstances.

PKI has been proved to be an effective and secure infrastructure in distributed environment. A subject owns a public/private key pair. The public key can be disclosed to anyone and the identity of the subject is represented by the public key. Certificate is used to transfer information between subjects. It is protected by the digital signature and can be disclosed to anyone.

We then examined a number of mainstream distributed access control models that are based on PKI. In their work, delegation is the most commonly adopted mechanism for decentralised administration of trust. The concept of delegation is that a subject (delegator) is able to grant a subset of his privileges to another subject (delegatee) therefore the delegatee could be able to access to specific resources. The deficiency of delegation is that the identities of both the delegators and delegatees and the details of the delegated privileges must be known in advance, which is sometimes impossible. This deficiency severely hampers the decentralisation of trust.

The main contribution of this thesis is our new model of decentralised administration of trust: *trust empowerment*. We tried to solve the limitations of delegation through this new model. In our model, trust is defined as properties and conveyed in certificate. The certificate is employed to help convey and establish trust between subjects. A certificate can also prove the credibility of other certificates. A subject is called *root of trust* if it is trusted by other subjects through its

identity. A subject only knows a limited number of *roots of trust* because of time and space restriction. *Trust empowerment* enables a subject to establish trust with many more subjects through their properties instead of identities. The access to the resource is determined by the certificates and policy. The features of *trust empowerment* are:

- Trust is subjective. Each entity sets its own trust policy which is independent of others. The consequence is that one entity is trusted at one place and may be untrusted at another.
- Fine-grained control. The service provider could finely control how the trust is conveyed from one subject to another and specify what each subject can do or cannot do.
- Trust is based on properties. Trust is defined as properties, which include attributes and capabilities. Properties can be owned and/or controlled. The property owner can use the privileges denoted by the property. The property controller can grant the property to other subjects but cannot use the privilege. A subject grants trust to another subject according to the attributes rather than the identities.

We have designed the Mojoy trust policy language that supports the concept of *trust empowerment*. The Mojoy trust policy language is partly based on XACML which is an OASIS standard. We have given the syntax, semantics and an XML implementation of the language. The advantages of using a language to specify trust are:

- Policy can be separated from the application and can be reused for different applications. One policy can even be distributed in many locations and updated uniformly.
- Certificate and policy share the same language syntax. Trust can therefore be administered in dispersed locations.
- A language is powerful and expressive enough to describe the complex situations such as advance RBAC, dynamic trust relationship, etc.

We have also developed a compliance checker for the language. The responsibility of the compliance checker is to tell whether a user is able to access the specified resource. The compliance checker takes in the requested action, user's certificates, local trust policy and the environmental parameters as input. It validates the certificates, looks through the policy and examines whether the request is supported by the certificates and allowed by the policy. The output is a Boolean value: Permitted or Denied, which indicates whether the user could perform the action on the resource or not. The compliance checker is developed in Java and is wrapped into

a JAR library and Web Service. It could be incorporated into any application developed in any language under any platform.

We have tried to apply our new trust model to a case study. The case study simulates a resource sharing circumstance between several organizations. These organizations offer resources to their own registered users. Several trust issues need to be addressed when these organizations want to share their resources with each other because the users are managed in different locations. We try to apply our *trust empowerment* to the situation and the model successfully solved all the trust issues. The achievements of our case study can be extended to practical healthcare domain.

7.2 Future work

Our work is only a very first step towards building a trustworthy computing network. Much more research is needed.

We have only provided a prototype of the language and more work is needed to complete the language. For instance, users might require using self-defined functions to complete complex tasks, which we do not support yet. Properties are divided into two categories in our design: attributes and capabilities. More sorts of properties are to be defined. For instance, property could be a program written by a subject to complete a complicated task. Subjects allow trusted peers' programs to be run under their working environment. The compliance checker currently only supports one language. Because of the possibility that different applications could adopt different languages, it should be able to accept the certificate and policy written in various languages. A possible solution is that the compliance checker accepts plug-ins to interpret various languages. Also, the present implementation of the compliance checker is only able to deal with one certificate and one policy. It is expected that future implementations will be able to process multiple certificates and policies.

Trust varies according to the context. Subjects establish minimal trust at the very beginning. They embark upon some primitive and inexpensive business. The trust level between them increases as the partners gradually accumulate experiences of each other. They will begin to do more and expensive transactions. This is analogous to the credit system widely accepted in modern economics. During the process, the trust policies remain unchanged. All completed transactions are logged and analysed. Trust level will increase/decrease according to the outcome

of the analysis of the transactions.

References

[1] M. Blaze, J. Feigenbaum, J. Ioannidis, and A. D. Keromytis, "The Role of trust management in distributed systems security", in Proceedings of Fourth International Workshop on Mobile Object Systems: Secure Internet Mobile Computations (MOS'98, Brussels, Belgium), no. 1603 in Lecture Notes in Computer Science, (Heidelberg, Germany), pp. 185-210, Springer-Verlag, July 1999

[2] W. Yao, K. Moody, J. Bacon, "A Model of OASIS Role-Based Access Control and its Support for Active Security", ACM Symposium on Access Control Models and Technology, SACMAT, Virginia, USA, May, 2001

[3] J. Bacon, K. Moody, W. Yao, "Access Control and Trust in the use of Widely Distributed Services", Proceedings of Middleware, 2001

[4] M. Blaze, J. Feigenbaum, and J. Lacy, "Decentralized trust management" in Proceedings of the IEEE Symposium on Research in Security and Privacy, (Oakland, CA), pp. 164-173, IEEE Computer Society, Technical Committee on Security and Privacy, IEEE Computer Society Press, May 1996.

[5] M. Blaze, J. Feigenbaum, J. Ioannidis, A. Keromytis, "The Keynote Trust-Management System", RFC 2704, September 1999, version 2

[6] D. Ferraiolo, and R. Kuhn, "Role-Based Access Control", proceedings of 15th national computer security conference, 1992

[7] R. S. Sandhu, E. J. Coyne, H. L. Feinstein, C. E. Youman, "Role-Based Access Control Models", IEEE Computer, Volume 29, Number 2, pp. 38-47, 1996

[8] Trusted Computer Security Evaluation Criteria, DoD 5200.28-STD. United States Department of Defense, 1985

[9] A. D. Keromytis, J. M. Smith, "Requirements for Scalable Access Control and Security Management Architectures", Columbia University Computer Science Department Technical Report CUCS-013-02, 2002.

[10] Walt Yao, "Trust Management for Widely Distributed Systems", PhD thesis, University of Cambridge, 2003

[11] D. W. Chadwick, O. Otenko, "The PERMIS X.509 Role Based Privilege Management Infrastructure", symposium on access control models and technologies, California, USA, 2002

[12] D. Chadwick, and A. Otenko, "RBAC policies in XML for X.509 based privilege management", in Proceedings of the 17th International Conference on Information Security, (Cairo, Egypt), 2002.

[13] "OASIS eXtensible Access Control Markup Language (XACML)", OASIS standard, Feb. 2003, version 1, http://www.oasis-open.org/committees/xacml/repository/

[14] N. Damianou, N. Dulay, E. Lupu, M. Sloman, "The Ponder policy specification language", policies for distributed systems and networks, International Workshop, (POLICY'01), Bristol, UK, pp. 18-38, 2001

[15] "The medical records confidentiality act of 1995", summary, http://www.cdt.org/privacy /medical/950000mrca_summary.shtm

[16] "A Brief Introduction to XACML", 14 March 2003, http://www.oasis-open.org /committees/download.php/2713/Brief_Introduction_to_XACML.html

[17] Andras Belokosztolszki, "role-based access control policy administration", Technical Report, Number 586, University of Cambridge, March, 2004

[18] J. Barkley, R. Bagwill, L. Carnahan, S. Chang, R. Kuhn, P. Markovitz, A. Nakassis, K. Olsen, M. Ransom, J. Wack, "security in open systems", NIST special publication 800-7, 1994

[19] D. F. Ferraiolo, D. M. Gilbert, and N. Lynch. "An examination of federal and commercial access control policy needs". In NIST-NCSC National Computer Security Conference, pp. 107-116, Baltimore, 1993

[20] M. Blaze, J. Feigenbaum, and M. Strauss, "Compliance Checking in the PolicyMaker Trust-Management System", In Proc. of the financial cryptography '98, Lecture Notes in Computer Science, vol. 1465, pages 254-274. Springer, Berlin, 1998

[21] M. Bartel, J. Boyer, B. Fox, B. LaMacchia, E. Simon, "XML-Signature Syntax and Processing", W3C, RFC3275, Feb 2002, http://www.w3.org/TR/xmldsig-core/

[22] T. Imamura, B. Dillaway, E. Simon, "XML Encryption Syntax and Processing", W3C, Recommendation, December 2002, http://www.w3.org/TR/xmlenc-core/

[23] Apache XML security library, http://xml.apache.org/security/

[24] Entrust Authority Toolkit for Java, https://www.entrust.com/developer/Java/

[25] VeriSign Java XKMS and XML Signature SDK, http://www.xmltrustcenter.org/xkms /developer/

[26] alphaWorks XML Security Suite, http://www.alphaworks.ibm.com/tech/xmlsecuritysuite/

[27] K. Chopra, W. A. Wallace, "Trust in Electronic Environments", Proceedings of the 36th Hawaii International Conference on System Sciences (HICSS'03), 2003

[28] "Authentication, Authorization, and Access Control", Apache HTTP Server Version 1.3, http://httpd.apache.org/docs/howto/auth.html

[29] The Open Group, "Authorization (AZN) API", January 2000, ISBN 1-85912-266-3

[30] ITU-T Recommendation, X.812 (1995) | ISO/IEC 10181-3: 1996 "Security Frameworks for open systems: Access control framework

[31] E. Barka and R. Sandhu, "Framework for role-based delegation models", in sixteenth annual computer security applications conference, New Orleans, Louisiana, Dec 2000

[32] Ezedin S. Barka, "Framework for Role-Based Delegation Models", PhD thesis, George Mason University, 2002

[33] R. Sandhu, "Engineering authority and trust in cyberspace: The OM-AM and RBAC way", in Proceedings 5th ACM Workshop on Role-Based Access Control (RBAC-00), (New York), pp.111-119, ACM Press, July, 2000

[34] J. Bacon, K. Moody, W. Yao, "A model of OASIS role-based access control and its support for active security", ACM Transactions on Information and System Security, vol. 5, Nov. 2002

[35] R. Hayton, "OASIS: An Open Architecture for Secure Interworking Services", PhD thesis, University of Cambridge Computer Lab, June 1996, technical report No. 399

[36] R. Hayton, J. Bacon, and K. Moody, "OASIS: Access control in an open, distributed environment", in Proceedings of IEEE Symposium on Security and Privacy (Oakland, CA, May), (Los Alamitos, CA), IEEE Computer Society Press, 1998

[37] E. C. Lupu, "A Role-Based Framework for Distributed Systems Management", PhD thesis, Department of Computing, Imperial College of Science, Technology and Medicine, University of London, 1998

[38] J. E. Tidswell, and J. M. Potter, "A graphical definition of authorization schema in the DTAC model", in Sixth ACM Symposium on Access Control Models and Technologies (SACMAT'01), pp. 109-120, 2001

[39] M. Sloman, "Policy driven management for distributed systems", Network and Systems Management, 2(4):333-360, 1994

[40] E. Bertino, S. Jajodia, and P. Samarati, "Supporting multiple access control policies in database systems", in IEEE Symposium on Security and Privacy (SSP'96), 1996

[41] W. Johnston, S. Mudumbai, and M. Thompson, "Authorization and attribute certificates for widely distributed access control", in Proceedings of the 7th IEEE International Workshops on Enabling Technologies: Infrastructure for Collaborative Enterprises (WETICE '98, Stanford, CA), (Los Alamitos, CA), IEEE Computer Society Press, June 1998.

[42] M. Thompson, W. Johnston, S. Mudumbai, G. Hoo, K. Jackson and A. Essiari. "Certificate-Based Access Control for Widely Distributed Resources". Proceedings of the 8th USENIX Security Symposium, Washington, D.C., 1999

[43] J. Bacon, K. Moody, J. Bates, R. Hayton, C. Ma, A. McNeil, O. Seidel, and M. Spiteri. "Generic support for distributed applications", IEEE Computer, pp. 68-76, March 2000

[44] UC Berkeley, "UNIX Programmer's Manual", 7 ed., 1981

[45] United States Department of Defense, "A Guide to Understanding Discretionary Access Control in Trusted Systems", NCSC-TG-003, National Computer Security Center, Sep 1987, http://www.radium.ncsc.mil/tpep/library/rainbow/NCSC-TG-003.html

[46] B. W. Lampson, "Protection", in Proceedings of the 5th Annual Princeton Conference on Information Sciences and Systems, (Princeton University), pp. 437-443, 1971.

[47] D. E. Bell and L. J. LaPadula, "Secure computer Systems: Mathematical Foundations and Model", M74-244, Mitre Corporation, Bedford, Massachusetts (1976). (Also available through National Technical Information Service, Springfield, VA, NTIS AD-771543.)

[48] K. J. Biba, "Integrity considerations for Secure Computer Systems", Mitre TR-3153, Mitre Corporation, Bedford, Massachusetts (1977). (Also available through National Technical Information Service, Springfield, VA, NTIS AD-A039324.)

[49] R. S. Sandhu, "Lattice-based access control models", IEEE Computer, 26(11):9-19, Nov 1993

[50] R. Yahalom, B. Klein, and T. Beth, "Trust relationships in secure systems – A distributed authentication perspective", in Proceedings of the 1993 IEEE Computer Society Symposium on Security and Privacy (SSP '93), (Washington –Brussels –Tokyo), pp. 150-164, IEEE, May 1993

[51] "URIs, URLs, and URNs: Clarifications and Recommendations", version 1.0, W3C/IETF URI Planning Interest Group, W3C Note 21 September 2001, http://www.w3.org/TR /uri-clarification

[52] T. Berners-Lee, R. Fielding, U.C. Irvine and L. Masinter, "Uniform Resource Identifiers (URI): Generic Syntax", Request for Comments (RFC) 2396, IETF (Internet Engineering Task Force), August, 1998.

[53] B. McLaughlin,"Java and XML", O'Reilly & Associates Inc., 2000

[54] L. A. Phillips, "Using XML", Special Edition, Que, Indianapolis, 2000

[55] World Wide Web Consortium recommendation, "Extensible Markup Language (XML) 1.0", Third Edition, http://www.w3.org/TR/REC-xml, Feb. 2004

[56] World Wide Web Consortium recommendation, "Exclusive XML Canonicalization Version 1.0", http://www.w3.org/TR/xml-exc-c14n/, July 2002

[57] "XML Schema Part 0: Primer", W3C Recommendation, May 2001, http://www.w3.org /TR/REC-xml

[58] "XML Schema Part 1: Structures", W3C Recommendation, May 2001, http://www.w3.org/TR/xmlschema-1/

[59] "XML Schema Part 2: Datatypes", W3C Recommendation, May 2001, http://www.w3.org /TR/xmlschema-2/

[60] ISO (International Standardization Organization) /ITU-T (Telecommunication Standardization Sector, International Telecommunication Union) Recommendation X.509(2001) The Directory: Authentication Framework

[61] P. NAUR (ed.), "Revised Report on the Algorithmic Language ALGOL 60.", Communications of the ACM, Vol. 3 No.5, pp. 299-314, May 1960.

[62] A. V. Aho, B. W. Kernighan, and P. J. Weinberger, "The AWK Programming Language", Addison-Wesley, Reading, 1988.