Access Rights Management with Delegation Certificates

Simon N. Kimani

LYNGBY 2002
EKSAMENSPROJEKT
NR. 00/99

IMM

Trykt af IMM, DTU

Abstract

A grid, which is essentially a large distributed system, is a software and hardware infrastructure that is an answer to the demand for high end computation capability. Grids are characterized by presence of a large pool of resources that span several domains and organizations. These resources need to be managed to ensure confidentiality, integrity and availability.

Due to the enormous size of these grids and certain security requirements unique to grids, existing security solutions are inadequate. This thesis makes an analysis of security requirements in a grid and specifies a policy that would meet the confidentiality security objective if enforced. We make an investigation of *SPKI* and *KeyNote trust-management engine* to come up with implementation proposals. These implementation proposals are compared to determine which of the two would be suitable for implementing the stated security policy. The SPKI implementation proposal that is recommended would be the basis of a prototype implementation.

Acknowledgement

I would like to thank my supervisor Professor Robin Sharp for coming up with the idea in this thesis and for his direction and insight during the project. It has been great working together.

Next I would like to thank my family back in Kenya that has had to miss me throughout the duration of my two year M.Sc study at DTU. I am glad that they understand the importance of this study to me.

Finally lots of thanks to my special friends John, Emma and Elly here in Denmark who made student life bearable by sharing with me many nice parties and special occasions that brought some cheer to student life. I look forward to more happy days with you guys.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

This chapter introduces the grid computing infrastructure with an intense focus on the grid security problem, section 1.1.4, which is the main interest of this thesis. After that we formulate a problem, section 1.2, whose solution will be a solution to the grid security problem.

1.1 Computation Grids

1.1.1 Motivation for Grids

The development of the grid has been motivated by the need to increase, both aggregate and peak, computational resources to important applications and to enable coupling of geographically separated people and computation resources to support collaborative engineering. The grid will find application in the following computing classes [47]:

1. Distributed Supercomputing : many grid resources are used to solve very large problems.
2. High-Throughput Computing : grid resources are used to solve large numbers of small tasks.
3. On-Demand computing : grids are used to meet peak needs for computational resources.
4. Data-Intensive Computing : grids are used to couple distributed data resources.
5. Collaborative Computing : grids are used to connect people thereby supporting communication and collaborative work between multiple partners.

These state-of-the art applications require large amounts of data and computation resources. These applications are used in modeling and simulating complex scientific and engineering problems, medical diagnosis, industrial equipment control, weather forecasting, biodiversity studies, planning systems for things such as traffic and other purposes.

There are various opportunities available to increase computation resources delivered to these applications. One of these is the computation grid [45], an idea motivated by the electric grid, a landmark innovation of the 20^{th} century [31].

1.1.2 Architecture

A computation grid is a hardware and software infrastructure that provides dependable, consistent, pervasive and inexpensive access to high-end computational capabilities [47]. We are interested in an internet type grid architecture. This is a grid which consists of a large pool of resources that transcend all kinds of boundaries (national, organizational, institutional e.t.c) supported by hardware interconnection and software to monitor and control these resources. Resources in this grid include storage systems, computer cycles, data archives etc.

This architecture is characterized by lack of central control and a wide geographical distribution of resources. These two make the grid complicated and challenging for performing network computing. When a grid spans several international borders, export controls may constrain technologies that can be used.

The grid consists of heterogeneous multiple trust domains[1]. Each domain is a collection of users and resources governed by a single administration and a single security policy. The administration determines the security requirements and ensure they are maintained across the domain. A domain can be an institution, organization, company e.t.c or a part of these such as a department. Domains are possibly hostile to one another and only cooperate to the extent that they need each others resources and services.

1.1.3 Research Challenges

Unique characteristics of the grid, described in preceding sections, have presented new problems that will have to be solved before the grid can develop as envisioned. These problems, from [47], include security, communication, resource allocation, resource location, process management and data access.

The primary interest of this report is the security challenge of the grid. The challenge of security in the grid is different from normal security concerns due to the large number of entities involved, stringent performance requirements, collective operation, downloading code and high resource sharing. A detailed description of the security challenge is given in section 1.1.4.

1.1.4 The Grid Security Problem

Grid applications are characterized by use of large number of resources located in multiple domains, dynamic resource requirements, complex communication and strict performance requirements. These resources are allocated dynamically during execution.

In a typical execution, a large computation may involve hundreds or less parallel processes that will acquire multiple-domain computation resources. These processes could in their lifetime start other processes that might also acquire resources dynamically. Processes constituting a computation may communicate by using a variety of mechanisms, including unicast and multicast.

Figure 1.1 is a rather contrived example of a typical grid application. In this sample grid application [49] a scientist in site A who is involved in a multi-institutional scientific collaboration needs to analyze data provided by site C using a program he has developed. So he sends code to site C with a request to run this program to analyze that data. This program starts and during its execution needs to run a simulation in order to compare the experimental results with prediction. To do this it contacts a resource broker service maintained by the collaboration (at site D), to locate idle resources that can be used for the simulation. The resource broker in turn initiates computations on computers at two sites (E and G). These computers access parameter values stored on a file system at another site F and also communicate among themselves (perhaps using specialized protocols, such as multicast) and with the broker, the original site and the user.

Above example highlights the following distinctive characteristics of the grid.

- User base is large and dynamic. Participants in a virtual organization like the one described are members of the institutions involved and change frequently.
- Resource pool is large and dynamic. Individual users and institutions decide whether and when to contribute resources. Resources are constantly added and removed from the grid.
- During its lifetime, a computation is composed of a dynamic group of resources running on different resources and sites.
- The processes constituting a computation may communicate using a variety of mechanisms, including unicast and multicast. Although these processes are logically the same, low-level communication connections (such as TCP sockets) may be created and destroyed severally during program execution.

[1] A *trust domain* is a logical, administrative structure within which a single, consistent local security policy holds [49].

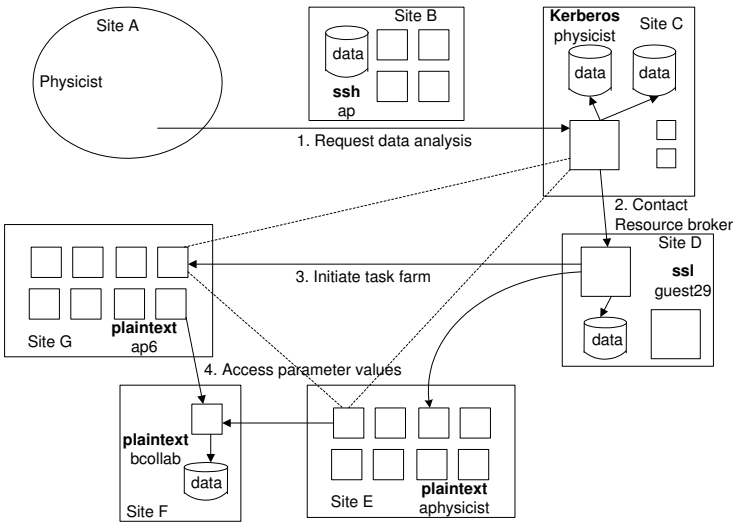

Figure 1.1: Example of a large distributed application: user initiates a computation that accesses data and computation resources in multiple sites.

- Resources may require different authentication and authorization mechanisms and policies in different sites.
- An individual user may be associated with different local name spaces, credentials, or accounts in different sites for the purpose of access control.
- Resources and users may be located in different countries.

The resources available in a grid are dynamic as domains continuously offer new resources and remove resources on offer. This necessitates a dynamic allocation of resources to processes. A resource *matching* service is required that can find the required type of resources in the grid. Optionally, the resource matching service should find resources that the processes are permitted to use.

1.2 Problem Statement

Problems with existing identity-oriented security mechanisms highlighted in section 2.6 have led to development of new discretionary access mechanisms that can achieve decentralized access control without relying on a Trusted Computing Base.

These are discretionary access control systems that are based on delegating access rights to public keys using public key certificates. Signing is done using public key cryptography. One key delegates access rights to another without dependence on trusted name and key services such as the X.500 directory. Cryptography keys are the entities. Any names that might be used are local rather than global [5].

Participants are all untrusted the way that computers in open networks are since these mechanism do not use any trusted third parties to certify entities. Instead of a hierarchical or centralized mechanisms of establishing trust, these systems provide ways of building local relations (trust) that are based on personal and business relations between participants. All keys can issue certificates to delegate access rights to resources they control. There is no central authority, such as TCB in identity-oriented systems, that controls the flow of access rights. Certificates and ACLs can list keys directly rather than identities.

This thesis will investigate proposals for new key-oriented mechanisms described. We will investigate two proposals for the use of delegation certificates for access control and compare the proposals to assess which one offers an efficient implementation in a large distributed system. The proposals to be investigated are *SPKI* [10] and *trust-management engines* [19].

1.3 Organization of the report

Chapter 2 reviews existing solutions for distributed systems. We first describe identity oriented systems (Kerberos, DSSA and TAOS) and then describe capability based systems (Amoeba distributed operating system). For each security system there is a description of the approach to security and limitations of using the approach. The objective of the chapter is to show the need for a new approach to security in distributed systems.

Chapter 3 has a description of the requirements for security in a grid. The security requirements identified and described are authentication, authorization, assurance, accounting, audit and secure communication.

Chapter 4 has a definition of the grid security policy that need to be enforced in utilizing protected resources in order to meet security requirements identified. The policy described should be enforced in order to meet the confidentiality security objective.

Chapter 5 describes the proposed architecture for grid security. This architecture describes how the security system is to be organized. There is a description of components in the system, their interaction and their function.

Chapter 6 describes how the proposed security system will inter-operate with existing security systems that are predominantly based on Kerberos security model.

Chapter 7 describes how SPKI certificates can be used enforce the grid security policy and related functionality requirements.

Chapter 8 describes how trust-management engines can be used to enforce the grid security policy and related functionality requirements.

Chapter 9 describes GSI, the current grid security architecture and a comparison with proposed security architecture.

Chapter 10 gives a summary of work done, describes some recommendations for further work and makes a conclusion of this thesis.

Appendix A contains a definition of key terms used in cryptography and a definition of TLS terminology.

Appendix B contains a definition of Kerberos terminology.

Appendix C contains a definition of SPKI terminology.

Appendix D contains a BNF definition of SPKI objects.

Appendix E contains a list of sample KeyNote assertions.

Chapter 2

Existing Security Mechanisms

2.1 Terminology

- A *principal* is a participant in a security operation. A principal can be a user, a process acting on behalf of a user or a process acting on behalf of a resource.
- An *object* is a resource that is being protected by the reference monitor using the security policy.
- A *credential* is a piece of information that is used to prove or establish some property (identity, authorization, delegation e.t.c.) of the holder, [49]. Passwords and certificates are examples of credentials.
- A *Digital certificate* is a statement signed (*certified*) by an entity (CA or principal e.t.c) which can be used to establish some property of the holder (such as identity, or accreditation, or authorization e.t.c.) [44].
- A *prover* is the entity that wishes access to a resource.
- A *verifier* The entity that processes request from a prover.
- A *reference monitor* is a guard for each object that examines each request for the object and decides whether to grant it.
- *Non-repudiation* is a service which prevents an entity from denying previous commitment or actions.
- *Data integrity* is a service which protects data against unauthorized manipulation by an attacker. Data manipulation includes such things as insertion, deletion and substitution.
- *Availability* is ensuring that legitimate users can have legal access to resources. To meet the availability requirement, a system has to be protected against denial-of-service attacks.
- *Access rights* The ability of a principal to use resources to perform operations on objects.
- *Access control* is the process of mediating every request to resources maintained by a system and determining whether the request should be granted or not [90]. The access control decision is enforced by a mechanism implementing regulations established by a security policy.
- *Confidentiality* is a service used to keep the content of information from unauthorized entities. There are numerous approaches to confidentiality ranging from physical protection to encryption that renders data unintelligible.
- *Delegation* is when one entity grants ability to acts on its behalf to another entity.
- *Digital signature* is a mechanism that enables a message recipient to convince a *third party* that the message as received originated from alleged sender.
- *Minimality* is a requirement that nothing is communicated to other parties except that which is specifically desired to be communicated. An example of an application of this principle is in the design of digital certificates so that they contain only the absolute minimum information to avoid revealing too much to other parties.
- *Authentication* is a process by which a principal proves its identity to a requestor, typically through the use of a credential. Authentication answers the question " Who said X ".
- *Authorization* answers the question " is X allowed to do Y on Z " for a principal X and some request

Y on some object Z.
- *Delegation certificate* is a digital certificate with which one entity delegates some access rights to another entity.
- *Users* are passive entities (persons) for whom authorization can be specified and who can connect to the system. After connecting to the system, users originate processes (subjects) that execute on their own behalf and submit requests to the system.
- *Ciphertext* The output of an encryption function. Encryption transforms plaintext into ciphertext.
- *Plaintext* The input to an encryption function or the output of a decryption function. Decryption transforms ciphertext into plaintext.

2.2 Kerberos

2.2.1 An Overview

Kerberos [77] is a network authentication service developed at MIT. Development began in 1985 with a release of five versions to date. Version 4 [92] was the first external as well as prominent version. The current standard version is V5 [60].

Kerberos provides authentication of users using the *trusted third party* authentication model [91] in the network allowing a secure access to resources on the network. Authentication is based on on DES keys, with each entity in the network sharing a private key with Kerberos authentication server. Knowledge of the secret key is proof of identity.

Kerberos provides a number of levels of protection. It provides mutual authentication at the start of a network connection, authentication for each message sent between entities, and optionally provides confidentiality and integrity of each message message sent between entities. Kerberos does not provide authorization services. Kerberos V5 can however pass authorization information generated by other services and in this manner be used as a basis for building separate distributed authorization services.

The Kerberos protocol is based on the Needham and Shroeder authentication protocol [75] but with some changes. These changes include addition of time stamps, addition of a ticket-granting server to support subsequent authentication without password re-entry, and a different approach to cross realm authentication.

2.2.2 The Model

Messages 1 and 2 are used to support a single sign on. The user logs in once supplying the password and subsequent authentication is automatic. When a user first logs in, an authentication request is issued (message 1) and a ticket and session key (message 2) for the ticket granting service is returned by the authentication server. This ticket, called a *ticket granting ticket*, has a short life typically on order of 8 hours. The ticket and session key are cached and the user's password is forgotten.

Subsequently when the user wishes to prove its identity to a new verifier, a new ticket is requested from the Ticket-granting service using the ticket granting exchange (messages 3 and 4). The client will authenticate itself using the ticket-granting-ticket and the request is encrypted using the session key issued. The client receives a ticket form the TGS that it can use in message 5 to authenticate to the verifier.

A *principal* is the Kerberos basic entity which participates in network authentication exchanges. Each principal is uniquely named by its principal identifier. A principal can be a user or a network service. The user's encryption key is derived from a password.

Multiple organization systems have multiple authentication servers each responsible for a subset of the users or servers. To prove its identity to a server in a remote realm, a Kerberos principal obtains a ticket granting ticket for the remote realm from its local authentication server. This requires the principal's local

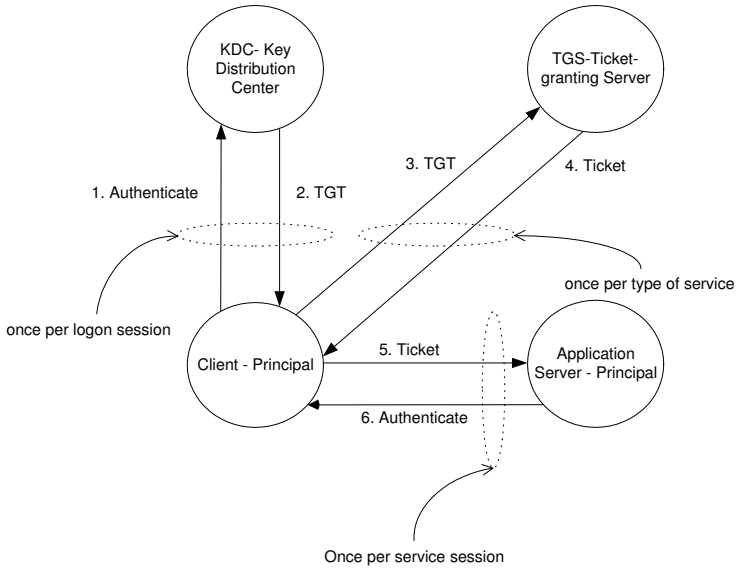

Figure 2.1: Kerberos

authentication server to share a cross-realm key with the verifier's authentication server. The principal next uses the ticket granting exchange to request a ticket for the verifier from verifier's authentication server, which detects that the ticket granting ticket was issued in a foreign realm, looks up the cross-realm key, verifies the validity of the ticket granting ticket, and issues a ticket and session key to the client.

Kerberos version 4 requires that each authentication server register with all other cross-realm authentication servers and share a unique key with each. In a network with N realms, this requires $O(N^2)$ keys to interconnect all realms. Version 5 supports multi-hop cross-realm authentication, allowing keys to be shared hierarchically. In this setup suppose there are two domains MIT.EDU and USC.EDU. Since they both share a key with EDU, they don't have to share a key with each for a principal, e.g bcn@MIT.EDU, in one domain to be able to prove identity to a verifier, e.g ptn@USC.EDU, in the other domain. This hierarchical organization of domains is similar to the hierarchical organization of certification authorities for public-key cryptography [35].

2.2.3 Limitations

Bellovin et al. [13] and McMahon [69] document the following limitations and weaknesses in Kerberos:

- It is possible to store and replay a ticket in its lifetime.
- Authentication relies on synchronization of clocks.
- Vulnerability to password-guessing attacks.
- Dependence on security of workstations.
- Since Kerberos depends on private-key cryptography, it does not scale well for large open distributed systems. Inter-domain interaction using public-key certificates would be a better solution.
- Kerberos does not provide non-repudiation services. This service requires public-key encryption.
- Kerberos provides only two security services for a distributed system: authentication and key exchange(for confidentiality and integrity). It does not provide authorization, delegation and services requiring public key technology. Due to this Kerberos does not scale well and is unsuitable for secure multi-domain distributed systems.

2.2.4 Extensions

Kerberos is recognized as a secure, tried and tested system that is widely used and a number of commercial packages based on it. Due to this fact there are a number of new systems that have been developed as

extensions to Kerberos that focus on eliminating the weaknesses.

Yaksha [53] is an extension to Kerberos that aims at adding public key facilities without changing the underlying protocol. The extensions focus on eliminating the following weaknesses in Kerberos that result from sole use of private-key cryptography: catastrophe from a compromise of KDC, dictionary attacks and lack of non-repudiation services. Yaksha uses an RSA variant, where two parties share an RSA key. Yaksha still lacks authorization and delegation and remains unsuitable for secure multi-domain distributed systems.

KryptoKnight was designed to provide the same set of services as Kerberos, but with a high degree of compactness and flexibility [87]. This is done by offering a more flexible protocol as well as using a subset of the Generic Security Service API (GSS-API) [66]. Kryptonight aims to provide the following services:

- Delegating of user identity to application programs.
- Mutual authentication between entities.
- Authentication of origin and integrity of exchanged data between entities.

This is done by offering four service classes

1. Single sign-on.
2. Two party authentication.
3. Key distribution.
4. Authentication of origin and content of data.

Kryptonight remains unsuitable for secure multi-domain distributed systems.

2.3 DSSA

2.3.1 An Overview

Distributed System Security Architecture (DSSA), [54], is a multi-domain architecture that provides single login, user and network entity authentication, mandatory and discretionary access control, secure initialization and loading, and delegation. The system is designed for a heterogeneous environment with no online security servers and no central control.

User authenticate themselves with Smart Cards containing private keys. Authentication is aided by the use of certificates, digitally signed by Certification Authorities and stored in a Distributed Name Service (DNS). A certification hierarchy exist. Principals that need to act on behalf of other principals are given the right to do so through certificates signed by delegating parties. Before software is executed on a computer, the MAC of the software is computed and compared against the software's certificate.

2.3.2 The Model

All entities in the system are called principals. Each principal has a security manager called a *reference monitor* with a local security policy. Reference monitors control access to principals that they maintain.

DSSA securely loads both operating system and application software. It does this by checking that the message authentication code (MAC) of the software image, is equal to the expected value for the software's specification. The MAC is a message hash, and the MACs of the software modules are are contained in certificates.

Each system, including hardware, holds a private key randomly generated when the system is created or installed. It uses this private key to authenticate itself and to certify systems it creates. DSSA defines a tree-structured DNS that is used to give global identifiers to principals. These DNS names are used in ACLs. DSSA uses the pull model to obtain privilege information. Groups have DNS names as well but they must exist as an explicit list of principals.

DSSA allows mutual authentication between a subject and an object. To deal with timeliness a challenge-response exchange is done at the beginning of each conversation. Mutual authentication results in a session key that can be used for further authentication and confidentiality. An online authentication is not necessary since signed certificates are used to authenticate public keys.

DSSA supports delegation as described in [55]. The architecture provides for a high degree of assurance that only authorized access is granted. Once access is revoked this no longer holds. Due to caching of access control information revocation is slow and is determined at application level. Some applications wait for next login while others wait for next access.

DSSA supports mandatory access as well through labeling mechanisms controlled by individual reference monitors. All objects and subjects have *access labels*, and access is enforced by the reference monitors.

2.3.3 Limitation

DSSA appears all perfect for the role for which it was created. The one identifiable architectural limitation is the use of off-line security servers. It is questionable whether off-line security servers are efficient and practical.

2.4 TAOS

A distributed systems is defined as a collection of nodes connected by an insecure network. Each node runs the Taos operating system [30]. Taos uses the access control model defined in 4.4 extended with compound principals. Compound principals are a uniform representation for sources of requests in a distributed system, including users, machines, channels, programs, delegations, roles, and groups.

In each node there is an OS component called *authentication agent* managing principals and their credentials. Applications access security services through an interface to the local agent. The agent implements all credential exchanges and validations, communicating with agents in other nodes when necessary and checking credentials according to the formal rules of logic stated in [6]. The agent uses a distributed certification database for names, group memberships and executable images.

Short term node-to-node channels are secured using using shared key encryption. All other encryption is public key, done for integrity and not secrecy. Authentication relies on signed *certificates*. Certificates are the building blocks for *credentials*. Certificates are valid for qualified time interval. Since certificates originate from different nodes, clocks at these nodes have to be synchronized.

2.4.1 Security Services

The authentication logic, defined in [6], used by Taos is too complex for direct presentation in a programming interface. To simplify it Taos defines a consistent set of security services. They are based on an abstract datatype **Prin** that represents principals, and a subtype **Auth** that represents principals that processes can speak for. Speaks for is a logic expression defined in the authentication logic. The security services are provided to applications in the form of an API. There are three interface types provided in the Taos API.

Authenticating messages: interface for sending and receiving authenticated messages. These are procedures that a process that can speak for a principal P can use to make another process believe P **says** S.

Basic authentication and authorization: This is the interface authenticating and authorizing requests.

Managing principals: The interface for managing Auths. Managing Auths involves changing the set of principals that the can speak. A Taos process can obtain an Auth in five ways:

- by inheritance from a parent process,

- by presenting a login secret,
- by a adopting a role,
- by delegating rights, or
- by claiming delegated rights.

2.4.2 Authentication Agent

The authentication agent handles the complexity of authenticating requests from compound principals. It has four parts: The secure channel manager that creates process-to-process secure channels; The Authority manager that associates Auths with processes and handles authentication requests; The credentials manager that maintains credentials on behalf of local processes and validates certificates authored on other nodes; The certification library that establishes a trusted mapping between principal names and cryptographic keys, and between groups and their members. Figure 2.2 shows the structure of the agent;arrows indicate call dependencies.

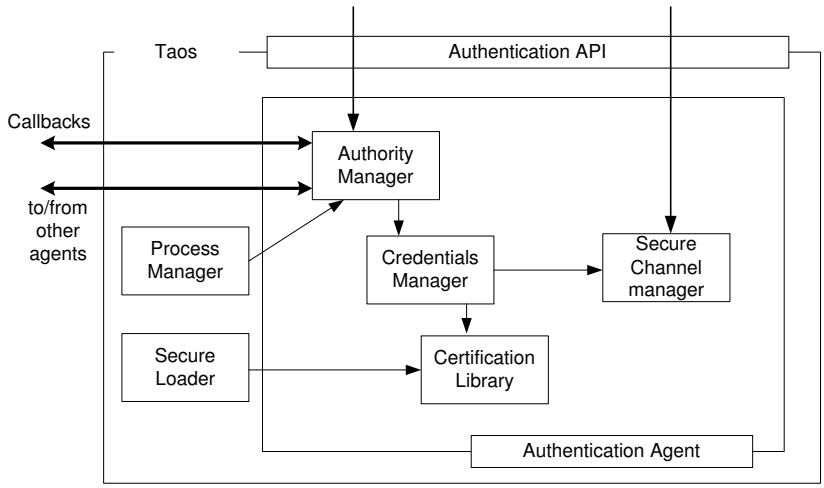

Figure 2.2: Structure of the authentication agent

1. The secure channel manager: It controls the construction of node-to-node channels and then utilizes these to provide process-level channels to clients. Since authentication is the process of proving that a channel utters a request on behalf of a principal, the secure channel manager must be able to attribute channels to processes and thereby link channels to the principals for which they speak. Methods for implementing secure channels are documented in [76].
2. The authority manager: The authority manager maintains a table with credentials for the Auths it creates. Each entry contains:
   - credentials for corresponding Auth,
   - a list of Ptags[1] of processes that own this Auth,
   - credentials for unclaimed delegations
   - a source from which to refresh delegation credentials

   The precise structure of credentials is irrelevant to the authority manager.
3. The credentials manager: The primary function of the credentials manager is to build, check and store credentials. Credentials have the logical meaning that one principal Q speaks for another principal P. Credentials have a grammar specified in [100, 14]. A certificate is an instance of a rule in the credentials grammar. Certificates have a valid time interval. There are different kinds of credentials:
   1. *Boot Certificates:* Certificates that describe a handoff from a machine key to a node key.

[1] A 32−bit integer used to identify a TAOS process. Unlike Unix process IDs, Ptags are never reused.

2. *Login and Session Certificates:* A login certificate is a special form of delegation certificate from a user's key to a conjunction of a node key and a temporary session key. To reduce the possibility of an attack, a user's key is in memory just long enough to sign the login certificate, a long duration certificate on the order of days. The node key and the session key are combined in a *session certificate*, which represents a handoff from the session key to the node key. A session certificate has a timeout and has to be refreshed repeatedly. At the end of a session the session key is discarded so that the session certificate can no longer be refreshed.
3. *General delegation certificates:* These general form of delegation transfer rights between principals.
4. *Channel certificates:* Since principals cannot utter requests directly, they handoff some of their rights to channels to act on their behalf. Requests on a channel are attributed to principals that handoff rights to them via channel certificates.

4. The certificate library ACLs contain names rather than keys. There is therefore a need to map keys to names at some point. Similarly there is need to map group names to group members and image digests to role names. This is the task of the certification library.
   Behind these services is a database. This database is the responsibility of the certification authority (CA). This CA is offline. Everyone trusts the CA and everyone knows the CA's public key. Certificates signed with this key are trusted. A user learns his own secret key and the CA's public key by decrypting a user-specific string stored in the name server. This string is DES encrypted using the users password. A smart-card could perform the same function.
   There are several kinds of certificates:
   1. *Name certificates:* These describe the mapping from keys to names. These are signed by a CA trusted for this purpose much like the CCITT X.509 certificates. These certificates are signed offline and can be trusted even if they are stored or transmitted in an untrusted way. Taos uses a replicated name service to store name certificates indexed by name.
   2. *Membership certificates:* These state that a principal U speaks for a group G.
   3. *Image certificates:* These are used in secure loading to verify the executable image of a recently loaded program and to name the role under which that program should run.

2.5 Capability-based Systems

2.5.1 Overview

Capabilities offer a unified mechanism for naming and protection. They are an alternative mechanism to Access Control Lists. In a capability-based system access to resources, hereby referred to as objects, is granted if the would-be accessor, subject, possesses a capability for the object. The capability is defined in [32] as a protected identifier that both identifies the object and specifies the access rights allowed to the holder of the capability. Capabilities can be passed between subjects and can also be increased or decreased in scope. Holders cannot however alter them without the mediation of the TCB.

A capability is like a "ticket" used to demonstrate that a holder has right of access to an object. It will contain the allowable access (read, right, execute e.t.c). They can be stored in files or in programs. They are protected using hardware or software or encryption mechanisms against direct modification by subjects. Processes can also pass capabilities to other processes. This is equivalent to delegation of access rights. This delegation however makes revocation difficult since it is not possible to determine all subjects that have access to an object. Revocation mechanisms are described in [39].

2.5.2 Amoeba distributed operating system

Amoeba is an object-oriented distributed operating system. It makes use of capabilities to provide a uniform mechanism for naming, accessing and protecting objects within the system [73]. To prevent users from tampering with capabilities they are protected using cryptography.

Amoeba has a client-server semantic model in which clients send messages to servers to manipulate objects managed by these servers. Servers have ports to which messages can be sent. Ports are 48 bit numbers only known to servers and their clients. Knowledge of port by a client implies that they are authorized to communicate with the server. Client however have to include capabilities in the messages they send to a server to prove they are allowed access to objects they want manipulated.

Clients and servers perform mutual authentication of one another. Clients are authenticated to the server through knowledge of the port that servers are listening on. Server authentication is however more complicated. It is done using a mechanism supported by a hardware device called the F-Box. The F-Box is put in the VLSI chip used for network interface and implements a mechanism for protection and security. The F-Box runs a secret function that is needed to understand messages sent using a corresponding client function. A software solution is also feasible using both conventional and public-key encryption.

Capability management in Amoeba is distributed. There is no monolithic capability manager. Each object is managed by some server which is a user process, rather than the kernel, that understands capabilities for its object. To create an object a client sends a request to the server specifying what it wants. The server creates the object and returns a capability to the client. On subsequent operations, the client must present the capability to identify the object.

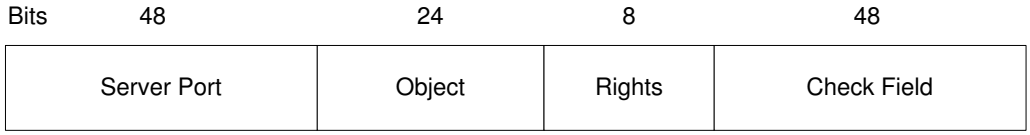

Figure 2.3: A capability in Amoeba

A capability has four fields: server port, object, rights and check. *Server port* is the port on which the server is listening, *Object* field is an identifier for the object, *Rights* fields is a bit map telling which operations are allowed to the holder of the capability. The *Check* is used for validating the capability. When an object is created, the server picks a random check value and stores it both in the new capability and inside its own table. Whenever the capability is passed back in a request for an operation, the check field value on the capability is verified using value stored in internal table.

In an **owner capability** all right bits are initially on. To create a new capability a client will pass back a capability along with a bit mask for the new rights. The server then creates a new capability and returns it to the caller. The calling process may send this new capability to another process to give rights to another process.

The capability mechanism is location transparent. To perform operations on an object, it is not necessary to know where this object is stored. This property provides support for dynamically changeable domains. In an insecure link it is necessary to add encryption to keep capabilities from being disclosed through wire tapping.

2.5.3 Limitations

Capability-based access control has been around for sometime but it is not in widespread use. Reasons for this include:

- Complexity of the mechanism make an efficient implementation challenging to achieve. It is difficult to revoke capabilities effectively, especially in off-line systems and especially when these capabilities have been delegated. This is because it is difficult to determine all capabilities that have access to an object and invalidate them.
- Capabilities are vulnerable to forgery. They can be copied and reused by an unauthorized third party.
- Dependent on TCB for new capabilities.
- Child processes cannot pass capabilities to one another. This is due to the fact that child processes are not aware of one another.

2.6 Shortcomings of Identity-Oriented Mechanisms

Access control problems in traditional security mechanisms can be divided into three sub-problems [29].

- Identification
- Authentication
- Authorization

Identification refers to the assignment of a unique identifier to each network entity so that the network entity is known to the system. Following successful identification and authentication, authorization links the identifier to a set of privileges and these are used to control access to the resource. One mechanism often used to specify privileges is the ACL. An ACL is a list describing what access rights a principal has.

These mechanisms rely on a trusted computing base (TCB). Users are uniquely identified to the TCB. This unique identifier is used by the DAC mechanism to perform authorization. Certificates are issued by a trusted or central authority. Access rights are given to names of entities and the names are separately bound to keys. This approach causes the following problems:

1. **Authentication**
   Determining the identity of principals depends on a centralized trusted authority such as in Kerberos V4 or a hierarchy of trusted authorities such as in Kerberos V5 or a hierarchy of certification authorities (CAs) such as in DSSA. Centralized trusted authorities are not scalable. Certification authorities use public key certificates. They are scalable but have other problems. PGP [102] and X.509 [96] are two commonly used certifying systems.
   The X.509 is one such scheme. The framework rests on the assumption that CAs are organized in a global "certifying authority tree" and for users to trust one another they need to possess certificates signed by CAs with a common ancestor in the global tree [34]. So who is going to be the root? Politics and sovereignty issues won't permit an answer to this question.
   Another problem is that it is not possible to implement a local trust policy such as have principal A trust principal B more than principal C. Trust is implicit and equal as long as principal CAs share a common root. This also causes the problem of "binary authorization", "all-or-nothing" access [19].
2. **Name spaces**
   Identity certification implies binding a name to a key. It is therefore necessary to choose a name space that gives globally unique identifiers. Two common naming schemes are X.509 and PGP.
   The X.509 certificates use a hierarchical naming scheme based on the X.500 directory service. It is designed in such a way that organizations can give unique names to their employees and other entities they control. Several years after it was designed it appears unlikely it will ever be fully implemented [28]. There is organization and individual privacy issue in publishing such a directory. The CIA would definitely be reluctant to publish a list of their agents. Individual names will also constantly change with change of employers. When these names change, the certificates become unusable even though they have not been compromised.
   PGP certificates use a flat name space with names of the form <Full name, EmailAddress>. What happens when individuals change names? This name space suffers some of the afflictions of the X.500 directory service.
3. **Delegation**
   Delegation is necessary for scalable distributed systems. It permits *decentralization* of administrative tasks. Delegation in ACL-based mechanisms is via ACL entries that are made by the TCB after certifying entity. Capabilities have an improved delegation mechanism based on issuing capabilities to delegates. Delegators send a request to the security manager managing the object, part of TCB, for new capabilities and then send these capabilities to delegates. This reliance on the TCB in both mechanisms is a bottleneck.
4. **Authorization**
   Identity-oriented certificate mechanisms create an indirection between certified information (answering the question "who is the holder of this key?") and the question that an application has to answer ("can this key be trusted for this purpose?") [20].

5. **Expressibility and Extensibility**
   Traditional ACL-based approaches do not provide sufficient expressibility and extensibility. Policy elements that cannot be expressed in ACL have to be hard-coded into applications. This approach is therefore neither scalable nor extensible.

In view of these problems, the solution is to have mechanisms that merge the three sub-problems (identification, authentication and authorization) into one and come up with a unified approach to solving the three problems. There has been such attempts and two such proposals are introduced in the coming section.

Chapter 3

Grid Security Requirements

A vital requirement for any system is to protect resources against unauthorized disclosure (*confidentiality*) and unauthorized or improper modification (*integrity*) and ensure availability to legitimate users (*no denials-of-service*).

Security requirements for traditional distributed systems remain applicable to the grid as well. Grids however introduce extra requirements of protecting applications and data from the systems on which parts of a computation will execute. Due to the fact that code comes from many different sources, there is also need to verify the origin and authenticity of code. Managing security in a grid is a challenge because resources are managed by many organizations often with different security requirements and possibly conflicting security policies.

To meet the primary goals of our security policy (which are confidentiality, integrity and availability) the following security requirements have to be met:Authentication, Authorization, Assurance, Accounting, Audit and Secure communication. This chapter describes these requirements in detail.

3.1 Authentication

Authentication is the process of verifying the identity of a participant to an operation or a request [45]. This participant is the *principal* on whose authority the operation is performed. Grid authentication requirements include [45]:

- *Mutual authentication:* of client and server is required. Client authentication protects servers from clients and server authentication ensures that resources are not provided by an attacker.
- *Data origin authentication:* is required to determine the principal from which a data item or executable object originated.
- *Delegation of identity:* is the process of granting authority to a principal A to act as another principal B. This is necessary when processes need to acquire a different identity to perform particular operations. Users must able to endow a program the ability to run on their behalf, so that the program is able to access the resources on which the user is authorized [44]. The program should (optionally) be able to re-delegate to another program.
- *Single login:* Users must be able to " log on "(authenticate) once and then have access to any resource in the grid that they are authorized to use [44].
- *Integration with various local security solutions:* Each site may employ any of a variety of local security solutions including Kerberos, Unix security, e.t.c [44]. The solution must be able to inter-operate with local solutions. It should not require wholesale replacement of local security. Instead it should allow mapping into local environment.
- *User-based trust relationships:* For a user to use resources from multiple providers, the security system must not require each of the resource providers to cooperate in configuring the security

environment. If a user has the right to use sites A and B, the user should be able to use both sites without requiring administrators in the sites to interact.

3.2 Authorization

Authorization is the process through which it is determined whether access to a resource is allowed. Grid authorization requirements include [45]:

- *Authorization by stakeholders:* Resource owners or stakeholders must be able to control which subjects can access the resource and under what conditions.
- *Program code verification:* When a user is providing code to run it is not sufficient to make decisions solely on the basis of who the user is. The code also needs to be verified before running. A way to do this is to check the checksum of the executable file against a digitally signed checksum.
- *Delegation of authority:* This is a means by which a principal A authorized to perform an operation can grant another principal B that authority. It is a more restricted form of delegation than delegation of identity. If need arises principal B should be able to re-delegate privileges. Delegations also have to be revocable.

3.3 Assurance

Assurance mechanisms enable a service requestor to decide whether a candidate system or service provider meets the requestor's requirements for security, trustworthy, reliability and other characteristics [45].

Assurance is a form of authorization used for validating the authority of the service provider. This authorization applied to computer systems is also referred to as *accreditation*. Examples of assurance in the grid:

- an ISO9001 certification indicating certain level of service and equipment quality.
- a certificate from a recognized body indicating that a program is virus-free.
- an organization certificate that indicates how much an organization can be trusted with data.

3.4 Accounting

Accounting provides the means to track, limit, or charge for consumption of resources in a system [45]. Accounting will support payment or barter for the use of computing resources. Tracking can enable application of resource quotas for fair allocation of available resources.

- accounting should be distributed so that quotas are applicable on any node.
- accounting servers should be distributed and scalable across domains.
- a settlement and clearing process between accounting servers in different domains would be appropriate to avoid compartmentalization of computing resources.

3.5 Audit

An audit function records operations that have been performed by a system [45]. Each operation is matched to the requestor principal. Audit is useful for backtracking if something goes wrong and to track breaches of security.

An *intrusion detection system* will analyze the audit data to find patterns that fit the profile of a system intrusion. To detect network attacks, the audit function should be distributed or audit records should be transmitted to a central location for each administrative unit.

3.6 Secure communication

Security services discussed support confidentiality and integrity of stored data. Full security requires that sensitive data produced or consumed by remote tasks be protected during transmission. This requires *integrity* and *confidentiality* of the communication. Grid requirements for communication include [44]:

- *Flexible message protection:* An application must be able to dynamically configure a service protocol to use various levels of message protection. These could vary from none, integrity only or integrity and confidentiality. Factors that determines this choice are sensitivity of messages, performance required, communicating parties and transmission infrastructure in use.
- *Support for various reliable communication protocols:* Besides the TCP protocol which is the most widely used communication protocol for the internet, security mechanisms must be usable with an assortment of other reliable communication protocols. This may be necessary in specialized environments.
- *Supports for independent data units (IDU):* Security mechanisms must support applications that require "protection of generic data (such as file or message)". Examples of such applications are email and streaming media.

Chapter 4

Grid Security Policy

Enforcing protection requires that every access to a system and its resources is controlled and that *all and only* authorized accesses can take place. This is the process called access control. An access control system is based on the following concepts:

Security policy: is the set of rules that define the security subjects (e.g users), security objects (e.g resources) and relationship among them.

Security model: it provides a *formal* representation of the access control security policy and its working. The formalization allows the proof of properties on the security provided by the access control system being designed [90].

Security mechanism: it defines the low level (software and hardware) functions that implement the controls imposed by the policy and formally stated in the model [90].

The security policy defined here covers confidentiality requirements expressed by sections 3.1, 3.2 and 3.5. We will define a policy which, if enforced, meets these security requirements. To express the policy in formal terms, we use the calculus for access control defined by Abadi et al. [6].

4.1 A Calculus for Access Control

4.1.1 Overview

The Calculus presented by Abadi et al. [6] is a logic language that we will use to describe access control lists, protocols and policies at an abstract level without concern for implementation.

Abadi et al. [6] consider principals that have human friendly names. Our interest is in principals that are identified by keys. When we refer to a principal A, we are referring to the principal represented by the public key pair (K_A^{-1}, K_A) because that is the way our principals are represented. There are simple principals and composite principals. Composite principals are constructed from simple principals using a number of connectives as follows:

- A∧B: A and B are cosigners. A request from A∧B is a request that both A and B make.
- A **as** R: The principal A in role R.
- B|A (pronounced B quoting A): The principal obtained when B speaks on behalf of A, not necessarily with proof that A has delegated authority to B. By definition B|A **says** s if B **says** A **says** s.
- B **for** A: The principal obtained when B speaks on behalf of A, with appropriate delegation certificates; B **for** A **says** s when A has delegated to A (A says (B|A ⇒ B **for** A)) and B|A **says** s

In order to define the rights of composite principals an algebraic calculus is defined so that one can express equations such as

$$(\text{B}|\text{C})\ \textbf{for}\ \text{A} \equiv (\text{B}\ \textbf{for}\ \text{A})|(\text{C}\ \textbf{for}\ \text{A}) \tag{4.1}$$

and then examine their consequences.

A modal logic extends the algebra of principals. In this logic, A **says** *s* represents the informal statement that the principal A says s. S can be a request (such as " read file *foo* ") or an assertion (" C's public key is K "). The modal logic is a basis for algorithms and protocols.

The logic also underlies a theory of ACLs. $\supset$ stands for the logical connective implication. A *controls s* is an abbreviation for (A *says s)* $\supset$ *s*, which expresses trust in A on the truth of s. An ACL for a formula *s* is a list of assertions of the form A *controls s*. If *s* represents a request to a server, then the ACL entry A *controls s* records the servers trust in A on s and therefore that A will be obeyed when it says *s*. If the meaning of *s* is clear, the ACL for *s* will simply be a list of principals trusted on *s*. If A $\Rightarrow$ B and B *controls s* then A *controls s* as well and therefore ACLs don't have to list all trusted principals. When B is listed, access should be granted to any A such that A $\Rightarrow$ B.

Channels are treated as any other principals. There is no formal treatment of encryption functions and the logic does not deal with problems associated with time stamps and lifetime. There is no explanation why a principal gets to believe that messages are fresh and not replays. Such detailed treatment is offered by logic such as the one proposed by Burrows et al. [25]

4.1.2 A Logic for Statements

Following is an inductive definition of statement.

- There is a countable supply of primitive statements p_1, p_2,...(e.g "read file *foo*").
- If s and s$'$ are statements then $\neg$s, s $\wedge$ s$'$ (s and s$'$), s $\supset$ s$'$ (s implies s$'$) and s $\equiv$ s$'$ (s is equivalent to s$'$) are statements.
- If A is a principal and s is a statement, then A **says** s is a statement.
- If A and B are principals, then A $\Rightarrow$ B (A **speaks for** B) is a statement.

$''$s means that s is an axiom of the theory or is provable from the axioms. There are four axioms for statements given in [64]. The intuitive meaning of $''$A **says** *s* is not quite that A has uttered the statement s, but rather that we can proceed as though A has uttered s. A can for example make A **says** *s* by sending *s* on a channel known to speak for A. A *controls s* stands for (A *says s)* $\supset$ s.

4.1.3 A Logic for Principals

The symbols A and B denote arbitrary principals, and usually C denotes a channel. There are two basic operators on principals, $\wedge$ (and) and | (quoting). We can get to understand their meaning from axioms that relate them to statements.

$$''(\text{A} \wedge \text{B})\ \textbf{says}\ \text{s} \equiv (\text{A}\ \textbf{says}\ \text{s}) \wedge (\text{B}\ \textbf{says}\ \text{s}) \tag{4.2}$$

(A $\wedge$ B) says something only if they both say it.

$$''(\text{A} \mid \text{B})\ \textbf{says}\ \text{s} \equiv \text{A}\ \textbf{says}\ \text{B}\ \textbf{says}\ \text{s} \tag{4.3}$$

(A | B) says something if A quotes B as saying it. We still need further proof to be sure that A is saying the truth.

$$''(\text{A} \Rightarrow \text{B}) \supset ((\text{A}\ \textbf{says}\ \text{s}) \supset (\text{B}\ \textbf{says}\ \text{s})) \tag{4.4}$$

Which gives us an informal definition of speaks for.

4.1.4 Handoff and Credentials

The following *handoff* axiom makes it possible for a principal to introduce new facts about $\Rightarrow$.

$$''(\text{A } \textbf{says } (\text{B} \Rightarrow \text{A})) \supset (\text{B} \Rightarrow \text{A}) \tag{4.5}$$

This implies that A has the right to allow any other principal to speak for it. There is a simple rule for applying this axiom: when you see A **says** s you can conclude s if it has the form B $\Rightarrow$ A. The same A must do the saying and appear on the right of the $\Rightarrow$, but B can be any principal.

From this axiom we get a theorem asserting that it is enough for the principal doing the saying to speak for the one on the right of the $\Rightarrow$, rather than being the same.

$$''((\text{A}' \Rightarrow \text{A}) \wedge \text{A}' \textbf{ says } (\text{B} \Rightarrow \text{A})) \supset (\text{B} \Rightarrow \text{A}) \tag{4.6}$$

This theorem, called the handoff rule, is the basis of our methods for authentication. Whenever we use it we say that A′ has hand off A to B. A final theorem deals with the exercise of joint authority.

$$''((\text{A}' \wedge \text{B} \Rightarrow \text{A}) \wedge (\text{B} \Rightarrow \text{A}')) \supset (\text{B} \Rightarrow \text{A}) \tag{4.7}$$

These two last theorems show how we can prove B $\Rightarrow$ A from the axioms together with some premises of the form A′ **says** (B′ $\Rightarrow$ A′). Such a proof together with the premises is called B's *credentials* for A. Each premise has a lifetime and, and the lifetime of the conclusion, and therefore the credential, is the lifetime of the shortest-lived premise. A complete listing of all axioms and therefore the assumptions of the theory are found in the appendix of [64]

4.2 Access Control Policy

4.2.1 Identification and Authentication

Principals are identified by a public key pair (K, K^{-1}). The private key K^{-1} is secret and it is always assumed to be controlled by the principal. When a principal is issuing a requesting it is speaking for the owner of the private key. This is expressed as: $K \Rightarrow K^{-1}$.

Authentication is the process of determining who said statement s. Our authentication is based on demonstration that the principal knows the secret key K^{-1}.

4.2.2 Trust in Third Parties

Sharing resources is based on trust among the principals. We don't have a central certification authority (CA) to tell principals what keys they can trust. We adopt a model of trust that is close to the PGP model of trust. In this model if a Principal A (K_A^{-1}, K_A) trusts a principal B (K_B^{-1}, K_B) based on some relationship between the two, A can issue a certificate to express this. Supposing that A and B are two scientists collaborating in a grid security project, A can express knowledge and trust in B by issuing a statement:

$$K_A \textbf{ says } K_B \textbf{ controls } s \tag{4.8}$$

s is the level of trust that A has in B. Borrowing from PGP our levels of trust are trusted, partially trusted, untrusted and unknown. If B makes a request for delegation of rights on a resource that A controls, A would delegate rights depending on the level of trust in B. Statement 4.8 is equivalent to the message encryption: Encrypt(K_A^{-1}, (K_B, trusted)). Granularity of trust levels can be defined to suit the application.

In a grid, the user base is large and we need to spread trust relations faster than direct individual expression of trust described can achieve. We therefore allow principals already trusted to act as introducers to other principals pretty much like the PGP model works. This makes out trust model transitive so that if A trusts B and B trust C then that implies that A trusts C. If a principal's introducers are partially trusted, the principal will require two such principals as introducers in order to get trusted.

4.2.3 Roles

Roles are used by principals to reduce their authority. A user may want to distinguish acting as administrator from running gaming software. If A is a principal and R is a role, A **as** R represents A acting in role R. A **as** R is a weaker principal than A. One way for A to express the fact that it is acting in role R when it says s is for A to make A **says** R **says** s. This idea motivates us to treat a role as a kind of principal and to define A **as** R to be A|R, so that A **as** R **says** s is the same as A **says** R **says** s. Because | is monotonic, **as** is also.

4.2.4 Roles and Groups

Many roles are related to groups. If G is a group, there may be a role G_{role} associated with it, so that a member of a group G can act in the role of member of G. A member A of several groups F, G, H, I..can select the privileges associated with one , say G, by adopting the corresponding G_{role}. Without this role, A matches any ACL with an entry F, but as A *as* G_{role} it does not. A *as* G_{role} matches any ACL with an entry G *as* G_{role} and if we adopt the rule G = G *as* G_{role} then A *as* G_{role} also matches any ACL with an entry G.

The calculus does not give a formal operator for relating groups and roles [6]. This has been suggested as a possible extension of the calculus. However, in terms of policy, there is a suggestion for relating the two. A principal may be allowed to act as *G-role* only if the principal is a member of G. Roles can be adopted freely and any A can speak in the role *G-role* with the identity *A as G-role*. If an ACL grants access to *G as G-role* and not to *A as G-role*, A can only be granted access if it is a member of G.

To Illustrate the group policy we take an example in which a scientist A (K_A^{-1}, K_A) creates a group called grid-collaborators represented by a key pair (K_g^{-1}, K_g). He issues:

$$K_A \textbf{ says } K_g \textbf{ controls } trust \tag{4.9}$$

Statement 4.9 is equivalent to the message encryption: Encrypt(K_A^{-1}, (K_B, trust)). If A would like to make B a member of this group then A first allocates itself permission to speak for the group through the statement.

$$K_g \textbf{ says } K_A \Rightarrow K_g \tag{4.10}$$

A can make this statement since A is in possession of K_g^{-1}. Next A makes the membership statement:

$$K_A \textbf{ says } K_B \Rightarrow K_g \tag{4.11}$$

4.2.5 Roles and Programs

In order to describe the inter-relationship between roles and programs we use a hypothetical emacs sample application in which P refers to application name emacs, I refers to the application executable file emacs.exe and D is a hash of emacs.exe using a hashing algorithm such as MD5.

Acting a certain way is the same as executing a program. Roles can therefore be equated to programs. Suppose node N runs program text or image I. How can a machine N express that it is obeying a program I. Node N can make N **as** I **says** s for statement s made by a process running the program image I. Suppose I is compressed to a digest D small enough to be used as a role. Such a digest can distinguish

one program from another as well as the entire program text, so N can make N **as** D **says** *s* instead of N **as** I **says** s.

Roles need names and we can say that the digest speaks for the role. D ⇒ P can now be used to express the fact that digest D speaks for the program named P. There are two ways to use this fact. The receiver of A **as** D **says** *s* can use D ⇒ P to conclude that A **as** P **says** *s* because s is monotonic. Alternatively, A can use D ⇒ P to justify making A **as** P **says** *s* whenever program D asserts s. Principals also need to convince others about the roles they assume. Since A **as** P is coded as A|P this is easy. To make A **as** P **says** *s*, A just makes A **says** P **says** *s* as we saw earlier, and to hand off A **as** P to some other channel C it makes A **as** P **says** (C ⇒ P).

4.2.5.1 Loading programs

A principal B requests node A to load program P. B needs the right to consume some resources on A. A makes a separate process *pr* to run the program, looks up P in the file system, copies resulting program image into *pr*, and starts it up. Note that there is implicit trust for A by B. B has to trust that B is indeed running P if it says so.

If A trusts the file system to speak for P, it hands off to *pr* the right to speak for A **as** P. Now *pr* is a protected subsystem; it has an independent existence and authority consistent with the program it is running. Because *pr* can speak for A **as** P, it can issue requests to an object with A **as** P in its ACL and the request will be granted.

If A doesn't trust the file system, it computes the digest of the program text and looks up the name P to get credentials for D ⇒ P. After checking this, it proceeds as before. Our policy is that nodes should not simply trust the file system. Nodes should use *image certificates* to verify the executable and ensure secure loading of programs. These image certificates can also be used to name the role under which the program should run.

For a program to be loaded, there should be an image certificate that states that:

$$\mathrm{D} \Rightarrow \mathrm{P} \tag{4.12}$$

P in our case is the name of the program such as emacs. It would be sufficient for the node CA to issue the certificate:

$$\mathrm{K}_{ca} \textbf{ says } (\mathrm{D} \Rightarrow \mathrm{P}) \tag{4.13}$$

but this will require that CA repeatedly issues similar certificates whenever a new release of P is made by user U. It is therefore convenient to delegate the ability to make such a certificate to U by:

$$\mathrm{K}_{ca} \textbf{ says } ((\mathrm{U} \mid \mathrm{P}\text{-}\textit{owner}) \Rightarrow \mathrm{P}) \tag{4.14}$$

where P-*owner* is a name associated with P (such as emacs-*owner*). If K_u is U's key, we can conclude:

$$(\mathrm{K}_u \mid \mathrm{P}\text{-}\textit{owner}\ ) \Rightarrow \mathrm{P} \tag{4.15}$$

This means that U can release a new version of P with digest D by signing an image certificate:

$$\mathrm{K}_u \textbf{ says } \mathrm{P}\text{-}\textit{owner} \textbf{ says } (\mathrm{D} \Rightarrow \mathrm{P}) \tag{4.16}$$

from which we conclude that

$$\mathrm{D} \Rightarrow \mathrm{P} \tag{4.17}$$

Image digests can be computed using any secure one-way function such as MD5 or SHA.

4.2.5.2 Booting

Booting a machine is similar to loading a program. The result is a node that can speak for M **as** P, if M is the machine and P the name or digest of the program that is booted. There are two differences. The machine is the base case for authenticating a system, and it authenticates its messages by knowing a private key K_m^{-1} which is stored in nonvolatile memory. Making and authenticating this key is part of installing the M.

During installation, M constructs a public key pair (K_m, K_m^{-1}). To be able to distinguish between M and the programs its booting, K_m^{-1} has to be protected from these programs by hiding this key until the next reset. When M (or rather the boot code of M) gives control to program P (normally OS) it is handing over all hardware resources of the machine. This has a number of effects.

- Since M is no longer going to be around to multiplex messages, it invents a key pair (K_n, K_n^{-1}) at boot time, gives K_n^{-1} to P, and makes a certificate K_m **says** $K_n \Rightarrow$ (M **as** P). The key K_n is the node key.
- M needs to know that P can be trusted with M's hardware resources. Its enough for M to know the digests of trustworthy programs, or the public key that is trusted to sign certificates for these digests. Our policy has adopted the later.

4.2.6 Delegation Semantics

Delegation is a primitive for secure distributed systems. It is the ability of a principal A to give principal B the authority to act on A's behalf. When B makes a resource request to a third principal, B will present credentials that demonstrate that A has delegated to B. In our delegation policy, an entity that has access rights with an option to grant them can delegate these rights to another entity fully or partially. The grant option is an axiomatic or basic right for owners of resources.

4.2.6.1 Delegating identity

This is the simplest form of delegation in which principal A delegates all rights to principals B. B will get access to all resources that A is allowed to access i.e access is granted to B as though A made a direct request. A can achieve this by issuing the statement A **says** B $\Rightarrow$ A.

4.2.6.2 Delegating rights

Delegation is expressed using the **for** operator. Intuitively, B is going to act on behalf of A. The basic axioms of **for** are:

$$''\text{A}\wedge\text{B}|\text{A} \Rightarrow \text{B } \textbf{for } \text{A} \tag{4.18}$$

$$''\textbf{for} \text{ is monotonic and distributes over } \wedge \tag{4.19}$$

To establish a delegation, A first delegates to B making

$$\text{A } \textbf{says } \text{B}|\text{A} \Rightarrow \text{B } \textbf{for } \text{A} \tag{4.20}$$

B|A is used so that B won't speak for A by mistake. Then B accepts the delegation by making

$$\text{B}|\text{A } \textbf{says } \text{B}|\text{A} \Rightarrow \text{B } \textbf{for } \text{A} \tag{4.21}$$

This explicit action by B is required because when B **for** A says something, the intended meaning is that both A and B contribute and therefore both must consent.

From these two we deduce that:

$$(\text{A}\wedge\text{B|A})\textbf{says}\ \text{B|A}\Rightarrow\text{B}\ \textbf{for}\ \text{A using (4.2), (4.20), (4.21);} \tag{4.22}$$

$$\text{B|A}\Rightarrow\text{B}\ \textbf{for}\ \text{A using (4.6) and (4.18)} \tag{4.23}$$

In other words, given (4.20) and (4.21), B can speak for B **for** A by quoting A. When timeout is used to revoke delegations, A gives (4.20) a fairly short lifetime and B must ask A to refresh it whenever its about to expire.

Illustration of delegation of rights with certificates

1. After mutual authentication, A issues a certificate to B under A's key. This certificate states that A has delegated some authority to B.
2. When B wishes to request r on behalf of A, no further interaction with A is needed. B can present A's certificate to C, along with the request K_B *says* A *says* r.
3. C has evidence that B has requested r on behalf of A. C consults the ACL for r and determines whether the request should be granted.

To present this in the given notation lets first assume that B *serves* A means that B is a delegate for A. The delegation certificate from A to B can be expressed:

$$K_A\ \textbf{says}\ (\text{B}\ \textbf{serves}\ \text{A}) \tag{4.24}$$

and replacing K_A with A, this is same as

$$\text{A}\ \textbf{says}\ (\text{B}\ \textbf{serves}\ \text{A}) \tag{4.25}$$

If C trusts A on this statement, C gets

$$((\text{B|A})\ \textbf{says}\ r)\wedge(\text{B}\ \textbf{serves}\ \text{A}) \tag{4.26}$$

From the theory, this means

$$(\text{B}\ \textbf{for}\ \text{A})\ \textbf{says}\ r \tag{4.27}$$

C can consult the ACL for r. If B|A appears in this ACL, that is, if B|A is trusted on r, then C believes r and access is granted. According to our delegation policy if A is the resource owner the right to delegate is automatic and r is granted if B is trusted on r.

4.2.7 Obligation policy

- Obligation policies allow monitoring of a system, and specify actions to be performed under given conditions. They provide the ability to respond to changing circumstances. Policies can then be used to prevent denial-of-service situations, and perform intrusion detection. **Auditing and accountability** can be handled via proper use of obligation policies.
- Obligations are disseminated to distributed automated management managers.
- Our obligation policy is restricted to audit that should enable intrusion detection. The audit policy should meet the audit requirements specified in section 3.5. Every access control decision is a proof using the request as a premise. Steps in the proof are justified by rules given in section 4.1. In our security system this complete proof is written in the audit and it is a complete account of what access was granted and why. Rejected access requests are also logged into audit trail for later analysis by an intrusion detection system to determine possible system attacks.

4.2.8 Revoking Delegation

Principals are responsible for delegations that they issue. Principals cancel delegated rights by issuing a revocation certificate to cancel delegation certificates. Revoking a delegation in a chain breaks the chain and all delegation certificates following the revoked delegation cannot claim any rights.

To illustrate the revocation policy, let there be a revocation service R that has something to say (confirmation) about a delegation made by A to B. We like to demonstrate that R will continue to confirm (B **for** A), thereby keeping the delegation valid, until after receiving a revocation certificate. Assume that A makes the statement.

$$\text{A } \textbf{says } ((\text{B}|\text{A}) \wedge (\text{R}|(\text{B } \textbf{for } \text{A}))) \Rightarrow \text{B } \textbf{for } \text{A} \tag{4.28}$$

Using this statement, A has delegated some rights to B and instructed or assumed R to confirm this upon request. B accepts the delegation by issuing.

$$\text{B}|\text{A } \textbf{says } ((\text{B}|\text{A}) \wedge (\text{R}|(\text{B } \textbf{for } \text{A}))) \Rightarrow \text{B } \textbf{for } \text{A} \tag{4.29}$$

Upon request, R is expected to issue the following statement unless instructed otherwise.

$$\text{R}|(\text{B } \textbf{for } \text{A}) \textbf{ says } (\text{B } \textbf{for } \text{A}) \Rightarrow \text{R}|(\text{B } \textbf{for } \text{A}) \tag{4.30}$$

These three statements and the rules of the calculus can be used to conclude:

$$\text{B}|\text{A} \Rightarrow (\text{B } \textbf{for } \text{A}) \tag{4.31}$$

The proof for this conclusion is presented in Myrvang [74].

4.2.9 Administration policy

Administrative policies define who is authorized to modify allowed accesses [90]. We use a decentralized administration policy based on *ownership* and *delegation*.

- *Ownership:* Each object is associated with an owner. This is usually the creator of the object. Owners can grant and revoke access rights on the objects they own. To grant access to an object, owners issue a delegation certificate and to revoke the access they revoke the delegation certificate.
- *Delegation* the owner of an object can delegate to other users the privilege of specifying authorizations, possibly with the ability to further delegate it. The owner of the object will issue a delegate certificate specifying the access rights and a *grant-option* for the rights. If the grant-option is true the delegate can re-delegate the rights given either in full or partially with a grant-option.

Decentralizing administration through the delegation of administrative privileges makes administration more scalable but complicates revocation since users can no longer keep track of who can access their objects.

4.2.10 Authorization policy

- Authorization policy defines rules for checking the validity of request for actions.
- An entity has an access right if it is mentioned in the policy or if the right has been delegated to it by another entity that has the right to delegate.
- When there are a number of successive delegations, these form a delegation chain. An entity can have access rights based on a chain of delegations. A valid chain of delegations begins with a policy entry and ends with the entity itself. Effective access rights in a delegation chain is an intersection of all rights in the chain. If an entity has two such delegation chains, the effective access right is the union of the effective rights of each of the chains.
- We take an ACL to be a set of principals, each with some rights to the ACL's object. The ACL grants a request A *says s* if A speaks for B and B is a principal on the ACL that has all the rights the request needs.
- To explain what has to be satisfied for access to be granted, first we define the ACL:

$$\begin{aligned} \mathit{ACL} &= \mathit{list\ of\ Entry} \\ \mathit{Requester} &= \mathit{Entry} \end{aligned}$$

$$
\begin{aligned}
\textit{Entry} &= \textit{conjunction of for-list} \\
\textit{for-list} &= \textit{Principal-in-Roles} \mid \textit{for-list } \textbf{\textit{for}} \textit{ Principals-in-Roles} \\
\textit{Principal-in-Roles} &= \textit{Proper-Principal} \mid \textit{Role} \Rightarrow \textit{Role} \\
\textit{Membership} &= \textit{list of Entry}
\end{aligned}
$$

- The request is granted if the requester implies one of the ACL entries.
- Each ACL entry is a conjunction of *for*-lists, and so is the requester. For the requester to imply an ACL entry, it must be that for each conjunct of the ACL entry there exists some conjunct of the requester that implies it.
- A *for*-list implies another if they are of the same length, and each principal in the roles of the requester implies the corresponding one of the entry.
- A principal in roles Q *as* R_1 as ... R_n implies another Q′ *as* R'_1 as ... $R'_{n'}$ if Q implies Q′ and for each R_j there exists R'_k such that R_j implies R'_k
- An atomic symbol P implies another atomic symbol P′ if there is a chain of assumptions $P = P_0 \Rightarrow ... \Rightarrow P_n = P'$

4.2.11 Caching credentials

Making an access control decision involves gathering information and then using an algorithm to grant access. The recommended security organization is one in which only the checking algorithm and the encryption mechanisms and the keys are in the TCB. We should be able to fetch digitally signed messages from un-trusted storage effectively excluding the storage and channels from the TCB. The system should be fail-secure meaning that failure of non-trusted components would only deny a request that would have been granted and not grant a request that should have been denied. We will use this idea when we are invalidating the cache and when we interpret an ACL that denies access to principals.

To make frequent operations fast, a cache is used to store the result of access control proofs that have been done. As an example if it has been proven that $A \Rightarrow B$ this result can be cached so that the proof does not have to be repeated next time it is required. When these entries become invalid they have to be removed from the cache. There are two approaches to do this. The first approach is to remember all the caches that contain the entry and notify them. **This has a very high cost.** The second is to limit the lifetime of the cache and then refresh the entry from the source if it is used after it expires. We adopt this approach. This approach requires a trade off between frequency of refresh (frequent is desirable but expensive) and the time it takes for cache entries to expire. Cache entries are first made when there is a miss and later refreshed whenever they expire. They can be discarded at any time without affecting the system.

Like other revocation methods, refreshing requires the source to be available. It is difficult to have a source that is both highly secure and available. The compromise is to have a combination of the two, one highly secure with long lifetime and the highly available with short lifetime, such that both need to agree to make information available. In the worst case, revocation is delayed.

4.2.12 Protecting Nodes and Protecting Code

Users in the grid need to use remote processing resources. A user needs assurance that the node offering the processing resource to execute this code is not a masquerader. Nodes need to check the programs sent to them and some assurance that they can trust this code. There is need to ensure that executables have not been modified to have some harmful effect such as through introduction of trojans. Figure 4.1 gives the code execution scenario.

Entities

user2 is the resource owner and CA on node2. user2 has permitted user1 to run application P on node2.

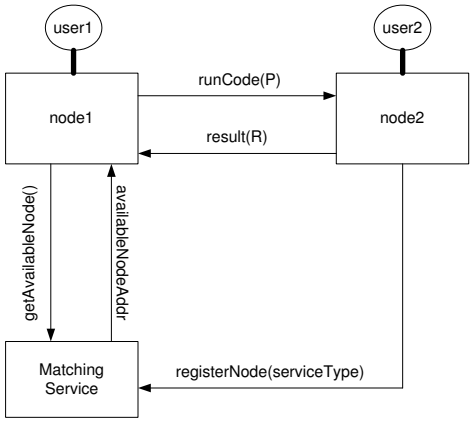

Figure 4.1: Code execution scenario in the grid

node2 is registered with the matching service as a service provider.

Matching service has a list of service providers. Nodes providing services register with the matching service. During registration they are authenticated to ensure they are not masqueraders or attackers. Clients that need services make a request to the matching service to get them an available instance of the service.

4.2.12.1 Protecting nodes

This descriptions uses the same emacs example given in section 4.2.5 In order to protect node2 from possible harmful code, user2 the node CA creates a group named trustedSW that all trusted software belong to. Only code belonging to this group can be executed on node2. D is the image digest for program P. For P to be executed on node2, it should have a membership certificate:

$$\text{D}\Rightarrow\text{trustedSW} \tag{4.32}$$

It would be sufficient for user2 to issue the certificate:

$$\text{K}_{user2}\ \textbf{says}\ (\text{D}\Rightarrow\text{trustedSW}) \tag{4.33}$$

but this will require that user2 repeatedly issue similar certificates whenever a new release of P is made by user1. It is therefore convenient to delegate the ability to make such a certificate to user1 by:

$$\text{K}_{user2}\ \textbf{says}\ ((\text{K}_{user1} \mid \text{K}_{user2})\Rightarrow\text{trustedSW}) \tag{4.34}$$

From this we can conclude:

$$(\text{K}_{user1} \mid \text{K}_{user2})\Rightarrow\text{trustedSW} \tag{4.35}$$

use1 can thereafter certify all releases of P by issuing the certificate:

$$(\text{K}_{user1} \mid \text{K}_{user2})\ \textbf{says}\ (\text{D}\Rightarrow\text{trustedSW}) \tag{4.36}$$

from which we conclude that

$$\text{D}\Rightarrow\text{trustedSW} \tag{4.37}$$

We refer to this as a **program trust certificate** for program P.

Every object that should be protected from an untrusted program gets an ACL of the form $(trustedSW)\wedge(\dots)$. The elided term gives the individuals that are trusted. To restrict access of resources only from trusted nodes the object gets an ACL $(SomeNodes)\wedge(\dots)$. Here *SomeNodes* is a group containing all the nodes that are trusted to access the object, and the elided term the trusted individuals. If node A cannot see a digest certificate for D the program then D⇒unknown and it can only access objects whose ACLs grant access to $(unknown)\wedge(\dots)$.

4.2.12.2 Protecting code

The matching service is responsible for protecting code by ensuring that only valid service providers are registered. In addition to this node1 and node2 have to mutually authenticate each other before the request to execute P on node2 is performed. This authentication should foil any server masquerade attempts.

4.3 Example

To illustrate the policy presented here is an example of an access request from Abadi et al. [6]. In this example a user A logged in a workstation B makes an access request to a server.

- The user A authenticates and delegates in some role R_A to a workstation B in role R_B. The user may use a smart card and use the delegation scheme described in section 4.2.6.

$$\mathrm{K}_A \textbf{ says } \mathrm{R}_A \textbf{ says } ((\mathrm{K}_d \wedge (\mathrm{B} \textbf{ as } \mathrm{R}_B)) \textbf{ serves } (\mathrm{A} \text{ as } \mathrm{R}_A)) \tag{4.38}$$

- The workstation sets up a secure channel to the server. This requires two statements:

$$(\mathrm{K}_d \textbf{ says } (\mathrm{A} \textbf{ as } \mathrm{R}_A) \textbf{ says } ((Ch \Rightarrow ((\mathrm{B} \textbf{ as } \mathrm{R}_B)) \textbf{ for } (\mathrm{A} \textbf{ as } \mathrm{R}_A) \tag{4.39}$$

  under the delegation key, and

$$\mathrm{K}_B \textbf{ says } \mathrm{R}_B \textbf{ says } (\mathrm{A} \textbf{ as } \mathrm{R}_A) \textbf{ says } (Ch \Rightarrow ((\mathrm{B} \textbf{ as } \mathrm{R}_B) \text{ for } (\mathrm{A} \textbf{ as } \mathrm{R}_A))) \tag{4.40}$$

  The workstation is working in the role R_B, a stronger role would work just as well.
- In our system the principal A is actually the key K_A and principal B is the key K_B. And using this property, if follows from the delegation certificate that

$$(\mathrm{A} \textbf{ as } \mathrm{R}_A) \textbf{ says } (K_d \wedge (\mathrm{B} \textbf{ as } \mathrm{R}_B)) \textbf{ serves } (\mathrm{A} \textbf{ as } \mathrm{R}_A) \tag{4.41}$$

  and this statement is believed. The channel setup certificate yields

$$((\mathrm{K}_d \wedge (\mathrm{B} \textbf{ as } \mathrm{R}_B)) \mid (\mathrm{A} \textbf{ as } \mathrm{R}_A)) \textbf{ says } (Ch \Rightarrow ((\mathrm{B} \textbf{ as } \mathrm{R}_B) \textbf{ for } (\mathrm{A} \textbf{ as } \mathrm{R}_A))) \tag{4.42}$$

  and this leads to

$$((\mathrm{B} \textbf{ as } \mathrm{R}_B) \textbf{ for } (\mathrm{A} \textbf{ as } \mathrm{R}_A)) \textbf{ says } (Ch \Rightarrow ((\mathrm{B} \textbf{ as } \mathrm{R}_B) \textbf{ for } (\mathrm{A} \textbf{ as } \mathrm{R}_A))) \tag{4.43}$$

  and then

$$Ch \Rightarrow ((\mathrm{B} \textbf{ as } \mathrm{R}_B) \textbf{ for } (\mathrm{A} \textbf{ as } \mathrm{R}_A)) \tag{4.44}$$

  as ((B as R_B) **for** (A as R_A)) is trusted in this matter.
- The user may adopt a further role R'_A in order to make a request r; or the workstation may do this on its own initiative.

$$(Ch \textbf{ as } R'_A) \textbf{ says } r \tag{4.45}$$

  The requester here is ((B **as** R_B) for (A **as** R_A)) **as** R'_A, which is the same as ((B **as** R_B) **for** (A **as** R_A **as** R'_A)).
- The ACL at the server may contain (G' **as** R'_B) **for** (G **as** R''_A). The server may have or the workstation may present certificates that prove that A $\Rightarrow$ G, R_A **as** R''_A, and B $\Rightarrow$ G'.
- Group certificates come with a signature. The signature's key has to be checked to ensure that it is trusted to certify group membership.
- After this the authorization algorithm can be used to determine whether access should be granted.

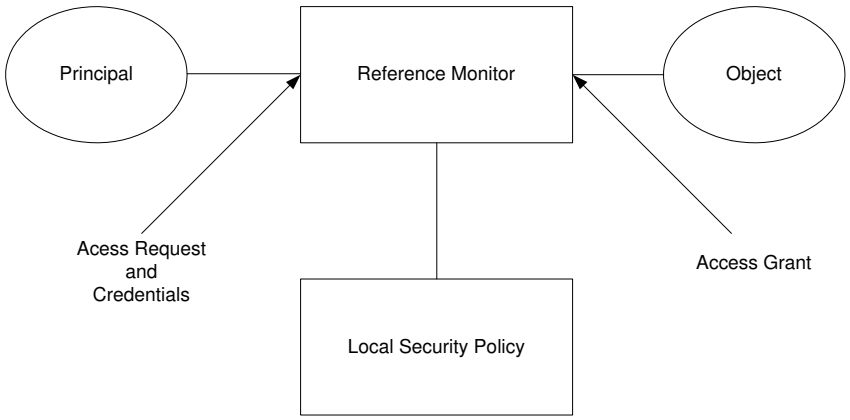

Figure 4.2: The access control model

4.4 Access Control Model

The reference monitor enforces all access mediation between principals and objects. To perform access control, a monitor needs to determine who the source of the request is, authentication, and then interpret the request to grant or deny access;authorization.

Access control policy enforcement requires two mechanisms: (1) is a mechanism to derive the access control policy associated with each principal (permission management mechanism) and (2) a mechanism to authorize controlled operations using this access control policy [58].

4.4.1 Permissions Management

Permissions management mechanism determine each principal's access rights at any time. Lampson [98] has formalized the representation of access rights into an *access matrix* with two practical variations: access control lists and capabilities. We are going to use a combination of the two. We will use ACLs to list the object owner, but rather than the traditional ACLs that list identities, these ACLs will list keys. To propagate these rights, principals will use delegation certificates. Delegation certificates are very similar to traditional capabilities except that possession of the certificate is not sufficient to permit access. Access is permitted by only those who posses the certificate and who can prove that they possess a cryptographic key named in the certificate. This overcomes the problem of forgery in capabilities.

Depending on the mechanism used to implement permissions used, the ACL entry can be replaced with a certificate as well. This is to take advantage of the fact that our ACLs will typically contain only one entry (object owner key) that is the anchor to the delegation certificates.

4.4.2 Authorization Mechanisms

Authorization mechanisms determine whether a principal is permitted to invoke an operation on an object. Logically, an authorizing agent retrieves the principal's access control policy and compares the request (perform an operation on an object) to the policy. If an entry in the policy grants the request and no policy precludes that request, then the authorizing agent permits the request.

Access control can be centralized or decentralized. A centralized model implements the traditional concept of a single monitor and a single security policy for the entire system [7]. This model is unsuitable for large distributed systems because it is not scalable. To solve this problem large distributed systems have decentralized security management. This security management model implements several monitors and several security policies. Each of these reference monitors manage access to a single object or groups of resources using a local security policy.

In present day systems, reference monitors are operating systems and large subsystems or servers that manage their own objects directly. In future any application could be a reference monitor for its own set of objects and subjects. We are going to assume a monolithic authorization mechanism in which the reference monitor is the operating system. The OS kernel stores all objects to which access is controlled and they have got access to the name spaces of the objects they need to protect. The kernel uses its internal representation of the system name space and its internal representation of the system security policy to authorize system calls. This approach is not suitable if (1) name spaces of objects to be controlled are outside the kernel. (2) applications need kernel to aid in enforcing their security requirements [58]. We will assume that these two do not arise.

Processes that belong to the same computation communicate freely. Communication between processes in different computations has to be controlled. For a client to invoke an operation on a server process, the client must have the right to send a message to a port to which the server has receive rights.

Chapter 5

An Architecture for Secure Grid Computing

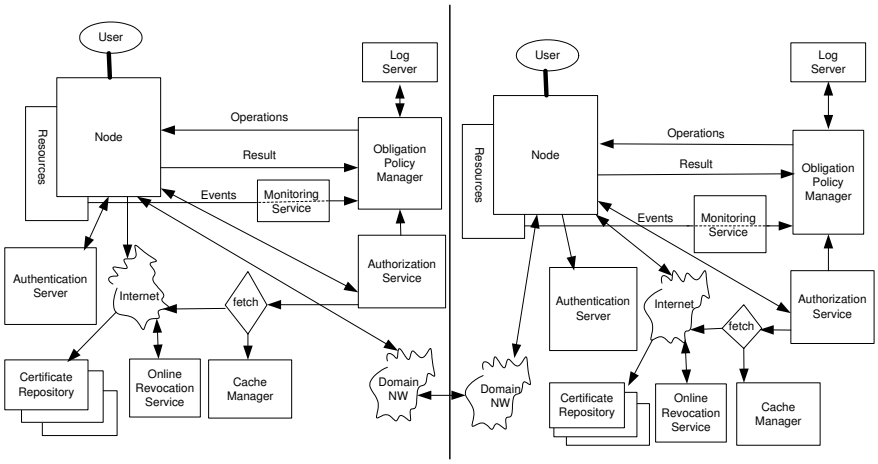

Figure 5.1: Security Architecture

The security policy defined in chapter 4 is the basis for constructing the security architecture shown by figure 5.1. Our architecture proposal comprises the set of entities in the security system and the interaction between them. These entities include subjects (users, processes, nodes), objects (grid resources) and application components (certificate repository, authorization service etc).

5.1 User(s)

5.1.1 Creating new users

PKI assigns to each user a matched pair of keys, one private and one public. A sender encodes a message by using the recipient's public key, then uses own private key to digitally sign the message. The recipient accesses the sender's public key to verify the signature, then uses own private key to decode the message.

The private key is private and must be kept secret. We propose here that the architecture support storage of private keys in PKCS[1]♯8 format encrypted with DES using PKCS♯5 password-based encryption.

[1]PKCS: Public Key Cryptography Standard

According to [97], one typical application of password-based encryption scheme is private-key protection, where the encrypted message contains private-key information, as in PKCS♯8. Using this encryption scheme, the private key can be stored at the Authentication Server.

The companion public key is public. Digital public key certificates, issued by "certification authorities" (CAs), are designed to distribute public keys and ensure their authenticity. Public access to public keys allow recipients of digitally signed messages to verify those signatures. It also allows coded messages to be sent between users who have never met. The architecture should support publishing and retrieving of these public key certificates in an LDAP directory.

Nodes in the grid will run a process that is responsible for creating new users. This node process makes use of a server process on the authentication server. There is a secure channel between these two processes used to send user information from the node to the server.

New users are required to provide a user−id and a password. This information is sent to the server. At the server the user-creation process will check that the user identifier is unique and then generate a unique key pair for this user. The server then makes a digital certificate for the public key. This certificate can contain user attributes if need be. Such certificates will be globally unique which is a required property of the system. This public key certificate is then published in a directory (LDAP directory) for distribution. Any entity that needs this public key will obtain it from this repository. The server will then store the private key pair, structured and encrypted as described earlier, indexed by the user−id for future retrieval upon request by this user.

What key should be used to sign the certificate? The admin key, the CA for the node, should be used to sign the certificate because the node installation key is not available to the OS and we don't want to use the temporary key K_n to sign the certificate because it is changing on every reboot.

5.1.2 User-to-Node Authentication

The login mechanism described here is motivated by Kerberos [62]. To authenticate to a node, users need to prove knowledge of a secret which in our case should be the private key. Users are however not good at remembering keys. Nodes therefore provide routines that can convert passwords into keys, making authentication the task of demonstrating knowledge of a password. Figure 5.2 shows the process of retrieving the private key from the authentication server. This process can be replaced with smart cards which we have not considered. Abadi et al. [4] has such a proposal.

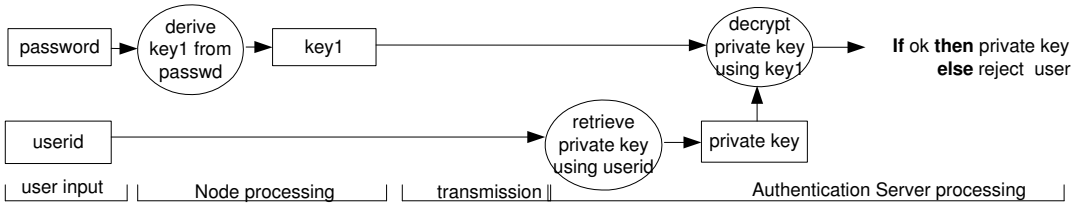

Figure 5.2: User key retrieval process

5.1.3 User-to-Node delegation

A variation of the basic delegation scheme, section 4.2.6, handles delegation from a user U to the workstation W during login. The assumption here is that the user's key K_u is available only during the login process. This is due to the intricacies that might be part of the login protocol such as supplying passwords or PIN numbers. The delegation of user to workstation therefore has a reasonably long lifetime to avoid perpetual refresh. To achieve this, the joint authority rule is used to require this delegation to have a counter signature by a temporary key K_l. This key is made at login time and is called a login session key. When the user logs out, the workstation forgets K_l^{-1} so that credentials that rely on this key can

no longer be refreshed after their 30-minute lifetime. If there is probability that workstation might be compromised within 30 minutes after logout, the node key should be discarded too.

The credentials for login start with a long-term delegation from the user to $K_w \wedge K_l$. K_w is the workstation's node key

$$K_u \textbf{ says } (K_w \wedge K_l)|\ K_u \Rightarrow K_w \textbf{ for } K_u \tag{5.1}$$

K_w accepts the delegation so we know that:

$$(K_w \wedge K_l)\ |\ K_u \Rightarrow K_w \textbf{ for } K_u \tag{5.2}$$

and since | distributes over $\wedge$ we get:

$$K_w\ |\ K_u \wedge K_l | K_u \Rightarrow K_w \textbf{ for } K_u \tag{5.3}$$

Next K_l signs a short-term certificate

$$K_l \textbf{ says } K_w \Rightarrow K_l \tag{5.4}$$

This enables us to make the conclusion $K_w\ |\ K_u \Rightarrow K_l|\ K_u$ using the handoff rule and the monotonicity of |. Now we can reach the conclusion for delegation with a short lifetime:

$$K_w\ |\ K_u \Rightarrow K_w \textbf{ for } K_u \tag{5.5}$$

5.1.4 Channel setup for user(s)

The node has the inverse of the public key and it can setup a channel Ch for $(K_l \wedge K_w)$ **for** K_u:

$$K_l \textbf{ says } K_u \textbf{ says } (Ch \Rightarrow (K_l \wedge K_w) \textbf{ for } K_u) \tag{5.6}$$

and

$$K_w \textbf{ says } K_u \textbf{ says } (Ch \Rightarrow (K_l \wedge K_w) \textbf{ for } K_u) \tag{5.7}$$

Hence

$$((K_l \wedge K_w) \textbf{ for } K_u) \textbf{ says } (Ch \Rightarrow (K_l \wedge K_w) \textbf{ for } K_u) \tag{5.8}$$

follows logically. When this statement is believed, it yields

$$(Ch \Rightarrow (K_l \wedge K_w) \textbf{ for } K_u) \tag{5.9}$$

and monotocity gives us the desired result:

$$Ch \Rightarrow K_w \textbf{ for } K_u \tag{5.10}$$

Thereafter K_w can make requests for K_u through the channel Ch.

5.1.5 Long running computations

Suppose a user starts a long running computation, typical of grid applications, that he would like to continue executing even after log off. The way to achieve this is to make a delegation to key K_l that lasts long enough to complete the computation. This has a risk of key subversion. To reduce this risk the use of this node should be restricted during the duration of the computation by only allowing trusted users to log into the node. The authentication server could be used to implement such a restriction. Lampson et al. [64] has another suggested solution that we have not considered. The solution is based on a long running agent representing the user that can renew expired credentials according to user instruction.

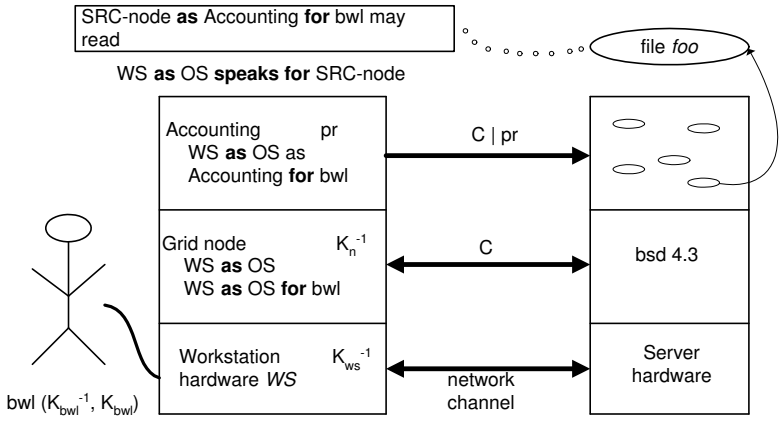

Figure 5.3: Principals and keys for a client-server example

5.1.6 Local user-to-process delegation

Figure 5.3 shows an example in which a user logs into a node WS and runs a subsystem that makes a request to an object implemented by a server on a different machine. The server must decide whether to grant the request.

For each component the figure indicates the principals that the component speaks for and the channel that it can send on. The node speaks for *WS* **as** *OS* and has the key K_n^{-1} so it can send on channel K_n. The accounting application runs as process *pr* and speaks for *WS* **as** *OS* **as** *Accounting* **for** *bwl*.

The node will setup a channel K_n|pr or C|pr on which the application can send requests. Now lets consider a request from the application to read file *foo*. The request has the form *C|pr* **says** "read file *foo*". *C|pr* is the channel carrying the request and it speaks for *WS* **as** *OS* **as** *Accounting* **for** *bwl*. The credentials of *C|pr* are:

K_{ws} **says** $K_n \Rightarrow K_{ws}$ **as** OS	From booting WS (Section 5.2.2)
K_{bwl} **says** $(K_n \wedge K_l) \mid K_{bwl} \Rightarrow K_n$ **for** K_{bwl}	From *bwl*'s login (Section 5.1.3)
K_l **says** $K_n \Rightarrow K_l$	Also from login
$K_n \mid K_{bwl}$ **says** C|pr $\Rightarrow$ $((K_{ws}$ **as** OS) **as** Accounting) **for** K_{bwl}	From channel setup for *pr*

To complete the access check, the server must obtain the group membership certificate *WS* **as** *OS* $\Rightarrow$ *SRC−node*. The scheme described is simplified so that a process speaks for only one principal.

5.1.7 Remote user-to-process delegation

Figure 5.4 shows an example in which a user has a login on node A. He needs to run an application but does not have enough local processing resource. He therefore runs this accounting subsystem on node B and it makes a request to an object implemented by a server on a different machine. The server must decide whether to grant the request.

For each component the figure indicates the principals that the component speaks for and the channel it can send on. Node A speaks for *A* **as** *OS* and has the key K_{an}^{-1} and it can send on channel K_{an}. The user can make requests on channel *C|bwl*. Node B speaks for *B* **as** *OS* and has key K_{bn}^{-1} and can send on K_{bn}. The accounting application runs as process *pr* and speaks for (B **as** *OS* **as** *Accounting*) **for** (*A* **as** *OS*) **for** *bwl*.

The node will setup a channel K_{nb}|pr or C|pr on which the application can send requests. Now lets consider a request from the application to read file *foo*. The request has the form *C|pr* **says** "read file *foo*". *C|pr* is the channel carrying the request and it speaks for (B **as** *OS* **as** *Accounting*) **for** (*A* **as** *OS*) **for** *bwl*. The credentials of *C|pr* are:

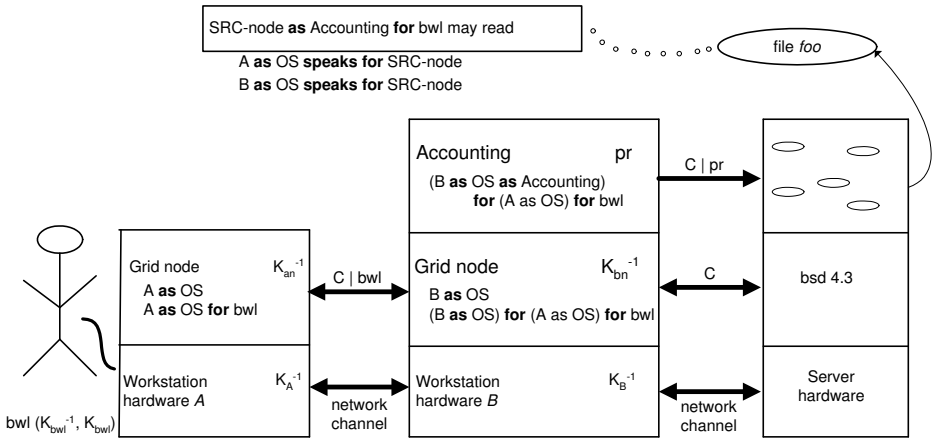

Figure 5.4: Principals and keys for a client-server example

K_A **says** $K_n \Rightarrow K_A$ **as** OS	From booting WS (Section 5.2.2)
K_{bwl} **says** $(K_{an} \wedge K_l) \mid K_{bwl} \Rightarrow$ K_{bn} **for** K_{bwl}	From *bwl*'s login (Section 5.1.3)
K_l**says** K_l**says** $K_{an} \Rightarrow K_l$	Also from login
$K_{nb} \mid (K_{an}$ **for** $K_{bwl})$ **says** C|pr $\Rightarrow$ $((K_B$ **as** OS) **as** Accounting) **for** (K_A as OS **for** $K_{bwl})$	From channel setup for *pr*

To complete the access check, the server must obtain the group membership certificates A **as** $OS \Rightarrow SRC{-}node$ and B **as** $OS \Rightarrow SRC{-}node$.

5.1.8 User-to-User delegation

5.1.8.1 Overview

We will adopt the TLS delegation protocol defined by Jackson, et al. [57] with certain modification to suit the grid delegation requirements. TLS delegation protocol is one of five client protocols in the TLS protocol. Section section 5.2.4 gives an overview of the TLS protocol. Reasons for basing our delegation on this protocol is due to certain attractions in the protocol:

1. The protocol allows the delegate and delegator to use an authentication protocol and authentication credentials of their own choice. The protocol a has data structure that supports X.509 certificates and Kerberos tickets. Authentication is an assumed property of the protocol and it therefore has no reliance on any particular mechanism.
2. The protocol allows the delegate and delegator to negotiate the delegation credential. This allows delegation of credentials different from the authentication credential. This will be useful when we cross authentication boundaries. Data structures already defined allow negotiation for X.509 delegation certificates (defined in draft−ggf−x509−proxy) and Kerberos 5 forwardable tickets (defined in RFC 1510). The data structures are going to be extended in chapters 7 and 8 with additional negotiable credentials. This section will simply outline the algorithm.

The TLS delegation protocol is specified for delegation of proxy−certificates. These proxy certificate are what we refer to as *delegation of identity* in section 3.1. In our architecture the equivalent of these identity delegations are user−to−process delegations, user−to−node delegations and node−to−node delegations. These delegations are performed differently. They are described in sections 5.1.6, 5.1.7, 5.1.3 and 5.2.3. Our delegation protocol here is used to pass delegation certificates, which are a kind of *pseudo-capabilities*, from one user to another.

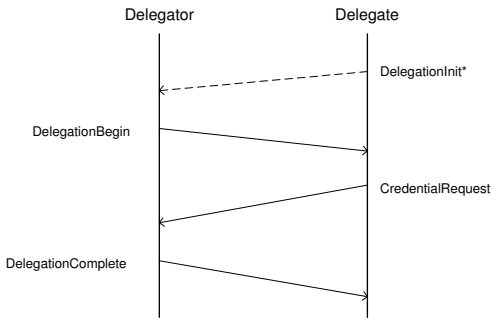

Figure 5.5: User-to-User delegation protocol

5.1.8.2 Protocol

Either the delegate or delegator may initiate the protocol. The delegate may send an optional DelegationInit message requesting the delegator to initiate the delegation protocol. DelegationInit contains the type of requested delegation credential. Delegates should not repeat the request until the delegation protocol is completed. The delegator may ignore the message if it is unwilling to perform the delegation at this time or respond with a no delegation error message if unwilling to perform delegation. DelegationBegin is used by the delegator to respond to DelegateInit from delegate or to initiate the protocol. The delegate then creates an algorithm specific delegation request and sends it in a DelegationRequest. The delegator responds to the request with the delegation credential in a DelegationComplete message.

Our delegation protocol delegates a variety of credentials (credentials for identity as well as object rights), unlike the TLS delegation that only delegates identity. We therefore need to modify the two initiation messages, DelegationInit and DelegationBegin, so that they specify what rights are requested or offered in the delegation. We use DelegationInit as a request by delegate for access rights and DelegationBegin as an offer of access rights to delegate. Both the request and the offer can be rejected using a delegation error message terminating the protocol.

5.2 Node(s)

Computers running an OS trusted for local security. The OS is part of the TCB.

5.2.1 Installing nodes

Certificates form semantically bound *chains*. These chains need a source of authority to anchor the chain. According to the administration policy in section 4.2.9 this should come from the resource owner. This same principle is used to create an initial security context during installation. By installation we refer to placing a node into operation.

A node M is represented by a key pair (K_m^{-1},K_m). In the installation process, this key is first created and stored in non-volatile memory. Next a pair of keys (K_{Admin}^{-1},K_{Admin}) is generated for the administrator doing the installation who also becomes the certifying authority (CA) for the node. Software executed on the node need an image certificate from this CA or from a user with a delegation to issue such a certificate as described in section 4.2.12.

The administrator installs the operating system (OS) and makes an image certificate for the OS.

$$K_{Admin} \textbf{ says } \text{D} \Rightarrow \text{P} \tag{5.11}$$

This image certificate is useful during the booting process, section 5.2.2. K_m makes a handoff to K_{Admin} to make the administrator the source of authority for the hardware resources on M.

$$K_m \textbf{ says } K_{Admin} \Rightarrow K_m \tag{5.12}$$

Resources added to the node later are owned by the users that add them and the users become the source of authority for these resources. However, since the administrator speaks for M, the administrator can take over these resources if the need arises. During installation or after the administrator can specify to the node who the authentication server is.

There is implicit trust of the node by the administrator that can be made explicit using a certificate.

$$K_{Admin} \textbf{ says } K_m \textit{ controls trust} \tag{5.13}$$

On completion of the installation, the administrator can boot the machine delegate resources to other users. The initial policy certificates that have been specified here are convenient to store locally.

5.2.2 Booting nodes

Booting a node is similar to loading a program. The result is a node that can speak for M **as** OS, OS being the operating system. The boot code for M is essentially handing over control of resources by passing control to OS.

We would like to distinguish the principal M from the weaker principal M **as** OS. To do this we hide K_m^{-1}. Since K_m^{-1} will no longer be available to multiplex messages, a new key pair is created (K_n^{-1},K_n) at booting time to act as the node key. K_n^{-1} is given to OS and a delegation certificate made.

$$K_m \textbf{ says } K_n \Rightarrow (\text{M } \textbf{as } \text{OS}) \tag{5.14}$$

Before loading OS, M checks the integrity of OS using the image certificate issued by its CA (the admin) described in section 5.2.1 to make sure that OS is trustworthy.

5.2.3 Node-to-Node delegation

The delegation described in section 4.2.6 is quite appropriate for node delegations. Here we demonstrate how to use it in cascade delegations. We suppose that node A has delegated to node B and that node B can operate with the identity B *for* A. If further delegation is required to node C, the delegation statements are as follows:

$$\text{B|A } \textbf{says } (\text{C } \textbf{serves } (\text{B } \textbf{for } \text{A})) \tag{5.15}$$

A has delegated to B and it follows that

$$(\text{B } \textbf{for } \text{A}) \textbf{ says } (\text{C } \textbf{serves } (\text{B } \textbf{for } \text{A})) \tag{5.16}$$

Believing this statement yields

$$((\text{C } \textbf{serves } (\text{B } \textbf{for } \text{A})) \tag{5.17}$$

Now C can make a request r on behalf of B for A

$$(\text{C } \textbf{says } (\text{B } \textbf{for } \text{A}) \textbf{ says } r) \tag{5.18}$$

and the delegation from A yields

$$(\text{C } \textbf{for } (\text{B } \textbf{for } \text{A}) \textbf{ says } r) \tag{5.19}$$

5.2.4 Secure channels

In the proposed architecture node-to-node channels are constructed using shared key encryption then multiplexed to provide process-level channels. Since the operating system has to be trusted for security, using encryption at a finer grain than this, for instance at process level, would not reduce the size of the TCB. Other techniques for implementing secure channels are well documented in [84].

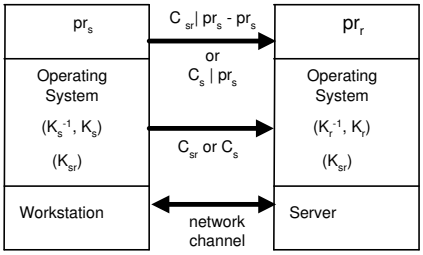

Figure 5.6: Communication Channels

Our proposal for *interprocess communication* allows authenticated requests to be sent from a sender to a receiver. The requests can be interpreted as one or more statements A **says** s. Requests use a channel between a sending process on a sending node and a receiving process on a receiving node.

The interprocess communication channel $C_{sr}|pr_s - pr_r$, from figure 5.6, is made by multiplexing a channel C_{sr} between the two nodes using the two process identifiers pr_s and pr_r as the multiplexing address. We do not investigate in detail how this multiplexing is done. The shared key K_{sr} is used to implement the node-to-node channel. We have seen in section 5.1.6 how the channel $C_{sr}|pr_s - pr_r$ acquires credentials from the user.

We propose to use TLS[2] [40] to setup the node-to-node channel. Setting up this channel implies getting the two nodes to share the secret key K_{sr}. TLS is a protocol based on the SSL 3.0 protocol specification published by Netscape. The primary goal of TLS is to provide privacy and data integrity between communicating applications. The TLS protocol is a two layer protocol. There is a low level protocol layer and a higher level client protocol layer. Figures 5.7 shows the constituent protocols of the TLS protocol.

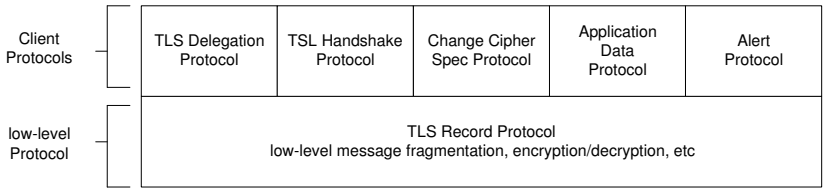

Figure 5.7: TLS Protocol

The *TLS Handshake protocol* is a suite of three client protocols used to allow peers to agree on security parameters for use by the record layer, authenticate themselves, instantiate security parameters and report error parameters. This protocol is responsible for negotiating a session (session identifer, optional peer certificate, compression method, cipher specification, master secret and a "is resumable" flag). Several connections can be instantiated using the same session through the protocol's resumptive feature. This is the suite of protocols that we use to setup the channels.

There are three protocols in the suite:

1. Change cipher specification protocol
2. Alert protocol
3. The TLS Handshake protocol itself used to establish cryptography parameters of the session. The client and server first agree on a protocol version, cryptography algorithms, optionally authentication each other and then use public key encryption techniques to generate a shared secret. Figure 5.8is an overview of this protocol. Refer to [40] for a detailed description of the protocol. Messages with a "*" or in brackets are optional.

The *TLS Record protocol* is the lowest layer in the two−layer TLS protocol. It sits on some reliable protocol such as TCP/IP. It a layered protocol that takes messages to be transmitted, fragments the data

[2]Transport Layer Security

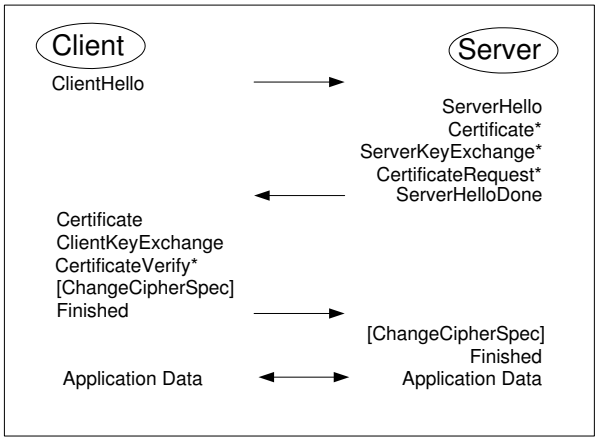

Figure 5.8: Message flow for a full handshake

into blocks, optionally compresses the data, applies a MAC, encrypts the data and then transmits the result. The reverse is performed on received data. The protocol has three vital components:

1. A connection state that defines the operating environment for the protocol. It specifies algorithms (compression, encryption and MAC) and their parameters (MAC secret, bulk encryption key and IVs). These security parameters are set by the TLS Handshake protocol. The security parameters for a TLS connection are set by providing the following values: connection end, bulk encryption algorithm, MAC algorithm, compression algorithm, master secret, a client random and a server random.
2. A Record Layer that receives uninterrupted data and performs fragmentation, record compression, record decompression and record payload protection.
3. A Key calculation algorithm that generates keys, IVs and MAC secrets from security parameters provided by the handshake protocol. The master secret is hashed into a sequence of secure bytes which are assigned to the MAC secrets, keys and non−export IVs required by the current connection state.

TLS supports has support for several block ciphers (DES, RC2, RSA and IDEA), stream ciphers (RC4), MAC (SHA−1 and MD5) and digital signatures (DSS and RSA).

5.3 Authentication Server

This server stores user key pairs indexed by their user identifiers. There should be at least one authentication server per domain. A domain in this context is a group of computers. Our group definition, described in section 4.2.4, is sufficient to represent domains.

The administrator for the authentication server node should define a group and issue membership certificates that can use the service. By doing this authorization for the service can be performed and it will additionally be possible to restrict users to particular nodes.

5.4 Authorization Service

The result of authentication by the reference monitor is a principal (e.g A, A **for** B **for** C **as** D) and a request (e.g read, write) for the resource. The reference will take these two and an ACL (e.g A *controls read*, A **as** B *controls write*) and ask the authorization service to prove that the principal speaks for one of the principals listed in the ACL entry. The authorization service returns a grant or deny for the request.

The authorization service will use existing proofs in the cache and credentials retrieved from the repository to perform the proof. The authorization is some kind of a daemon service that can be called by any reference monitor with the required premises. The formula proven and the result is passed to the obligation policy manager for posting into the log.

5.5 Cache Server

Proofs (such as A⇒B) that have been made are stored in the server with a validity time for later use. This minimizes the search for certificates especially for repeated requests. Some of the intermediate results in a proof can also be stored and used as lemmas in other proofs. If a proof has expired then an attempt to reuse it will cause a cache miss and necessitate a refresh of the cache entry.

Whenever we run out of space in the cache then an existing entry can be overwritten. This will not affect the working of authorization since whenever a proof does not exist in the cache, credentials have to be obtained from the internet certificate repository. The downside to the cache is that revocation can be delayed in a worst case scenario when a proof that has not yet expired is reused despite the fact that it no longer holds. Having a separate cache server would enable sharing of the same cache by several authorization services.

5.6 Certificate Repository

We need a distributed and fault tolerant directory to store certificates. Recently there has been proposals that allow digital signatures and certificates to be stored in the DNS [80]. The internet Domain Name System (DNS) [72] is a global directory system originally developed for storing and retrieving information about internet hosts.

The DNS naming space is a classical tree structure consisting of arcs and nodes. Nodes have labels (hierarchical names) and also contain data which is arranged as typed resource records (RR). Resource record types define what kind of data can be stored in the DNS, IP-addresses are an example. Creating new resource records require an IETF standard.

The DNS system is focussed on naming entities (hosts, services, user e.t.c) and storing their attributes. To store information for an entity, it has to be named in the DNS system. Our entities (users and nodes) are represented by keys. We can use a one way hash function (such as MD5) to create a hash string from the key and use it as the DNS name component. The IETF DNS Security (DNSSEC) working group has defined a DNS security standard [3] that specifies three resource record types. One of these, the public key record (KEY) type is used to attach keys to keyholders.

To store entity certificates in the DNS structure a new resource record has been defined by [2]. It is a single certificate record type able to contain any kind of certificate (X.509, SPKI, PGP plus future certificates). The CERT record format consists of four elements: type, key tag, algorithm number and the certificate or certificate revocation list. The key tag is a 16 bit hash of the subject key. The tag is used to tie related KEY and CERT records. Algorithm numbers are assigned by IANA and 1 has already been allocated to MD5/RSA. The certificate or CRL is stored in a BASE64 encoded string making the internal structure of the certificate invisible to DNS.

5.7 Obligation Policy Manager

Obligation policies defined are translated into TCL (Tool Command Language) script which is downloaded to automated managers. Lupu et al. [67] have specified a language that can be used to specify policy and check it for consistency. This policy specification can be converted automatically to TCL script and downloaded to the obligation policy manager. This approach will simplify update of the manager.

5.8 Log Server

All requests for resources and their results are logged into the log server. The results of the authorization server , both those denied and those granted, are passed to the obligation policy manager that logs them. Data from the log server is then analyzed by an intrusion detection system. The log server should be some kind of distributed storage that can replicate data to a central storage that is analyzed to assess resource usage within a domain and possible security breaches.

5.9 Monitoring Service

The monitoring service receives events from resources and acts on these events. Some of these events require policy related actions such as performing maintenance on resources. These kind of events are passed to the obligation policy manager. One other task that can be performed by the monitoring service is informing human managers about the occurrence of events via facilities such as paging, email e.t.c. The interest here is in the events that require duties to be performed by the obligation policy manager.

5.10 Revoking Delegations

We are going to use two kinds of certificates and each has a different revocation mechanism.

1. *Short-lived certificates:* These certificates expire after a given duration and need periodic refreshing for continued use. Its the kind of certificate that users use to delegate rights to nodes to act on their behalf. Revocation of these certificates is automatic after the duration of validity has expired.
2. *Long-lived certificates:* These certificates can be described as permanent in comparison to the previous ones. They require an explicit revocation. To revoke them we will use timed revalidations, a method of revocation motivated by SPKI [43]. Timed revalidation lists will be provided by an online service that can be queried about the status of a certificate. A revalidation list contains a certificate (or list of certificates) that is valid. To reduce the cost of revoking certificates, an issuer of a revokable certificate should explicitly state this on the certificate. The revocation service will be distributed, not centralized, and each certificate should have a reference to the server and signature key that provides the revalidation list. Revoked certificates have to be reported to the revocation service by the parties revoking them.
   Every authorization computation must be deterministic. To ensure this, SPKI has set three rules:
   (a) Certificates must list the key that signs the revalidation list and the location where list is to be fetched, in our case the online revocation service.
   (b) Revalidation list must have a validity date.
   (c) The validity dates must not intersect. One list can only be issued after expiry of another.
   No revocable certificate can be processed without a valid revalidation list making the list a complement to the certificate rather than a "surprise" announcement of change of mind.

5.11 Security system vulnerabilities

- The Authentication Server storing the keys has to remain secure since breaching its security would compromise security of the whole system.
- The security system is amenable to dictionary attacks. It is therefore imperative that users select appropriate passwords. The authentication server could provide a service that checks the suitability of passwords before accepting new ones.

Chapter 6

Inter-Operability with Kerberos

Kerberos is an authentication systems based on conventional cryptography. It is described in detail in section 2.2. The problem and solution discussed here is based on Kerberos Version 5 [61]. This version contains new features required for practical delegations described later in section 6.4.

Kerberos is the most widely used authentication system, with several commercial systems based on it. We therefore want to be able to add Kerberos-based domains to the grid if organizations that own such domains are unwilling to adopt the domain security management proposed in chapter 5.

6.1 Authentication in Kerberos

Kerberos credentials consists of two parts: a ticket and a session key. The ticket contains the principal's name and a session key. The session key is a secret shared between the client and the end−server. The ticket is encrypted in the secret key of the end−server. The secret key is encrypted in the session key shared by the client and the Kerberos server (AS or TGS).

To prove its identity to an end−server, a client sends the ticket to the end−server along with an authenticator encrypted using the session key. The authenticator serves to prove knowledge of the session key and avoid possible replay attacks.

The ticket and authenticator have a field named authorization. This field has an arbitrary number of typed sub−fields which can be used to place restrictions on the use of the ticket. Interpretation of these fields is open. Only one rule applies, restrictions are additive. When tickets are requested, requesting principals can specify restrictions on their use. Clifford [78] has a suggestion of possible implementation.

The first ticket acquired by a client has to be a ticket for the ticket-granting service (TGS). This ticket is called a *ticket−granting ticket* and is obtained from the authentication service (AS). Using this ticket, a client can obtain tickets to communicate with end servers from the TGS.

By establishing "inter-realm" keys, administrators of two realms allow a client authenticated in one realm to use its authentication remotely. Exchange of inter−realm keys (separate may be used in each direction) registers the ticket−granting service of each realm as a principal in the other realm.

A client is then able to obtain a ticket−granting ticket for the remote realm's ticket−granting service from its local realm. When that ticket−granting ticket is used, the remote ticket−granting service uses the inter−realm key to decrypt the ticket−granting ticket, thus authenticating the ticket, and then issue a ticket for the end−server. This ticket will indicate the clients realm since end−server may want to authorize such clients differently.

We are going to consider non-hierarchical realms to reduce the complexity of our problem. In this context, one realm is said to communicate with another realm if the two realms share an inter−realm key.

6.2 Using Kerberos credentials in TLS

Medvinsky et al. [70] extends TLS to include Kerberos cipher suites and Mathew et al. [56] extends this further to include delegation of Kerberos credentials in the TLS Handshake protocol.

The two document proposals put together achieve mutual authentication, establishment of a master secret and delegation of Kerberos credentials using the TLS Handshake protocol. With support for delegation of Kerberos credentials one client identity can enable end to end authentication within an n−tier architecture.

The basic TLS Handshake protocol remains the same as shown in figure 5.7. However, since authentication and the establishment of a master secret will be done using the client's Kerberos credentials for the TLS server, the optional messages server Certificate and ServerKeyExchange are omitted. The handshake protocol showing only messages required to use Kerberos credentials is outlined in figure 6.1

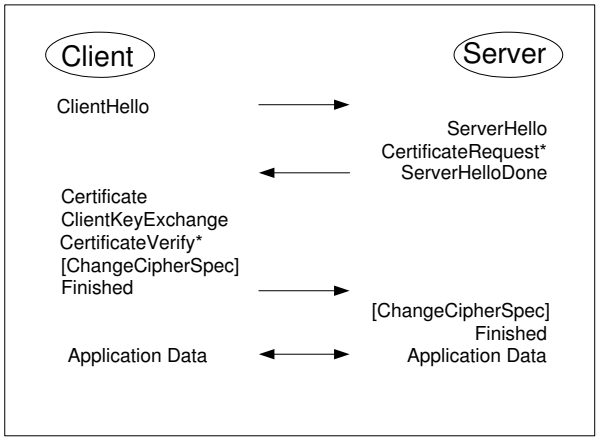

Figure 6.1: Message flow for a full handshake

Below follows an explanation of the use of each of the messages exchanged.

1. ClientHello message
   This message is used by the client to negotiate a security context. Client passes a list of cipher suites in order of client preference. Kerberos authentication requires a list of Kerberos cipher suites. [70] defines a number of Kerberos cipher suites. An example of a suite defined by [56] is TLS-KRB5-WITH-RC4-128-MD5 (meaning initial authentication using Kerberos V5, RC4 will be used with a 128 bit key and MAC is based on MD5 algorithm).
2. ServerHello message
   This message is used by the server to negotiate a security context. This message returns the cipher suite selected by the server if it is able to find an acceptable set of algorithms. If a match does not exist, the server will respond with a failure alert.
3. CertificateRequest message
   If server has accepted a Kerberos−based cipher suite, then it must send this message to convey Kerberos specific characteristics such as the realm. The TLS client is going to use this information to acquire a service ticket for the TLS server. If user−to−user authentication is required, the server has to send a TGT as well. The TGT is passed to the TGS in the ticket request. The TGS will use the session key from this ticket to encrypt the service ticket.
   This message is unauthenticated and enables the server to request a particular client principal as well as a particular service principal. In the event that a service principal name is specified there is a risk that the client may be tricked into requesting a ticket or delegating to a rogue server. To be sure that service ticket is obtained for the right server the client should use a combination of its own local policy, configuration information and information supplied by the KDC.
4. ServerHelloDone message
   This message is used by the server to indicate to the client that the ServerHello and associated

massages have ended.

5. Client Certificate message
   A client must respond with a certificate message after receiving a CertificateRequest message. A Kerberos certificate structure consists of a Kerberos AP-REQ message used to authenticate client to the server and an optional series of Kerberos KRB-CRED messages to convey delegated credentials. The client may use a local policy and other input from the TGS to decide on delegation.
6. ClientKeyExchange message
   This message is used to pass a random 48−byte pre−master secret to the server. The pre−master secret is used to independently derive the master secret, which in turn is used to generate the session key and for MAC computations.
   To use Kerberos authentication option, the TLS client must obtain a service ticket for the TLS server. With Kerberos authentication, the pre−master secret remains the same. It is encrypted using the session key and sent to the TLS server along with Kerberos credentials obtained. Once the ClientKeyExchange message is received, the server's secret key is used to unwrap the credentials and extract the pre−master key.
7. Client and Server Finished messages
   These two messages verify that the key exchange and authentication processes were successful. They are protected with the algorithms that have been negotiated.

6.3 Delegation credentials in Kerberos

Kerberos supports delegation of rights using proxy tickets and delegation of identity using forwarded ticket−granting tickets.

6.3.1 Proxiable and Proxy tickets

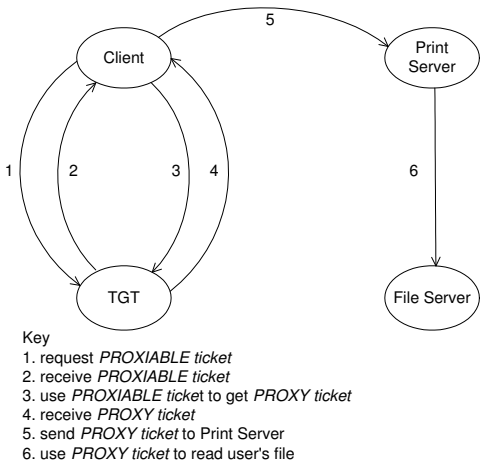

Figure 6.2: Client has given print server a proxy ticket to access client's file on the file server to satisfy print request

The IP address (or list of IP addresses) of the client is encoded inside every Kerberos ticket. This information is used by application servers and the TGS to verify the address of the client. By default a ticket acquired on one host cannot be used on another. A client can however request for a ticket from the TGS with the *proxiable flag* set. The client can use this ticket to acquire another ticket from the TGS that is tenable at another address. This second ticket is called a *proxy ticket* and has the proxy flag set. The *proxy ticket* is sent to a machine or service to give it the client's identity in order to perform

an operation on its behalf. Use of the identity is limited to a particular operation. Figure 6.2 shows the process of acquiring and using proxiable and proxy tickets.

6.3.2 Forwardable and Forwarded tickets

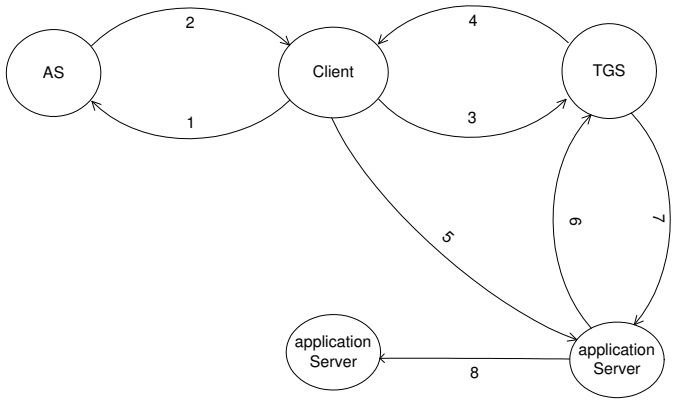

Figure 6.3: Application Server gets a client's identity to access all resources required to satisfy client request

These are tickets used for authentication forwarding in which the delegate (proxy) is granted complete use of the clients identity. The client obtains a *forwardable TGT* from the AS. To pass its identity to another principal or service, a clients makes a request for a *forwarded TGT* and passes this to the delegate. The delegate can use the *forwarded ticket* to obtain tickets from the TGS to communicate with end−server using the clients identity. Figure 6.3 shows the process of acquiring and using forwardable and forwarded tickets.

6.4 Delegation protocol

The delegation protocol defines how the delegator passes delegation credentials to the delegatee. The result is the possession of a *proxy ticket* or *forwardable TGT* by the delegate. There are two ways of doing this.

1. using the TLS delegation protocol described in section 5.1.8.
2. using the TLS Handshake protocol described in section 6.2.

6.5 Authorization

Our inter-operation with Kerberos is meant to support existing domains that use security systems based on Kerberos security model. The assumption is that these existing systems have an authorization mechanism based on authenticated Kerberos principals.

6.6 Proposed solution

What we need to achieve is get a non−Kerberos domain to communicate with a Kerberos domain. Kerberos is based on a trusted third party that is responsible for vouching for client identity. We need a similar entity in our domain and the AS is best placed to do this. Another requirement is that the two domains need to exchange keys as an expression of trust in each other so that local authentication in one can be used remotely in the other. To achieve this our AS will exchange keys with the TGS server in Kerberos domain. This way our nodes can obtain a TGT for the remote TGS and use the TGT to request a service ticket. The other alternative to this would be for each of the nodes to exchange a key with the Kerberos TGS server. For a large number of nodes this would require too many keys and we go for the former.

Authorization in our domain is dependent on certificates and identities. After a Kerberos client has authenticated to a node using an identifier, we still cannot authorize the request. To tackle this, each principal in our domain that trusts a Kerberos principal has to make an identity certificate for the principal to bind the identity to a pair of keys. The node will generate the key pair to which all rights are delegated using delegation certificates. These certificates will be the basis of authorization and we need not change the access control algorithm.

Figure 6.4 shows how a node in a non−Kerberos domain can access resources in a Kerberos domain and figure 6.5 shows the reverse scenario.

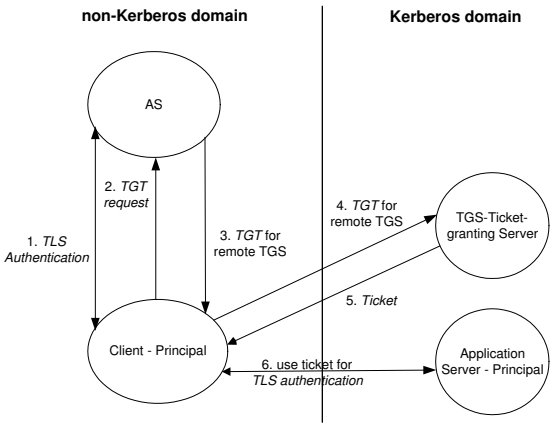

Figure 6.4: non-Kerberos to Kerberos realm operations

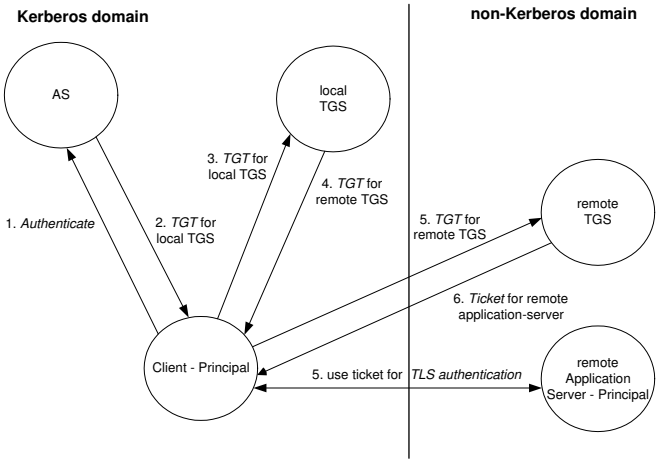

Figure 6.5: Kerberos to non-Kerberos realm operations

6.7 Shortcomings in the inter−operation

Kerberos does not support composite principals that is required to fully meet our access control policy. The most important of these composite principals are principals in roles. Kerberos treats principals and principals as roles the same way. Due to this there are some security checks that cannot be performed on Kerberos domains. Kerberos nodes are therefore less secure. This is due to fact that we retain the rather old existing authorization already running in these domains.

As an example our policy requires that decisions on access of resources be based on both the principals and the program that this principal is running and even the node that the user is logged on. Such a check would not be enforceable on a Kerberos because we cannot pass a composite principal to convey this information.

6.8 Inter−operability to PK−Kerberos

PK−Kerberos refers to Kerberos that supports authentication using public key cryptography. In PK-Kerberos, principals possess a public key pair that they use to mutually authenticate and establish a shared symmetric session key. Brian et al. [95] explains how to make this extension to Kerberos V 5. When Kerberos principals are public key there is a choice of either using TLS or "native" Kerberos authentication protocols defined in [95].

Sirbu et. al [68] has a comparison giving the pros and cons of using SSL 3.0 versus Kerberos public key authentication protocol. SSL 3.0 has the advantage of widespread support in the market place making it a formidable alternative to Kerberos authentication protocol for public key based principals. We therefore choose to use TLS (which is synonymous to SSL 3.0) over native Kerberos authentication protocol. This is a better option for us since Kerberos principals behave like non-Kerberos principals when they are public key based and we are already using TLS elsewhere.

Chapter 7

Grid Policy Management using SPKI/SDSI 2.0

We have stated our grid security policy in chapter 4. To enforce this policy we need an enforcement mechanism. Our enforcement mechanisms are digital certificates. This chapter will investigate the use of SPKI[1]/SDSI[2] certificates as the enforcement mechanism in the grid. In our description we will code name SPKI/SDSI to SPKI.

In order to use SPKI as the enforcement mechanism we need to encode the policy using SPKI certificates. We therefore go through the policy stated and encode the policy in SPKI. When principals create certificates to express policy, we end up with certificates that need to be managed. They need to be stored, retrieved and at times traversed to make conclusions such as authorization. This constitutes certificate related functionality that is depicted in the architecture. We will investigate what kind of algorithms, libraries e.t.c have been implemented to do this.

We assume that we have access to all the algorithms that we need for generating all kinds of keys, digital signatures and hashing. Below is a description of SPKI focussing on concepts that we use in the grid. A complete description can be found in [43] and [42].

7.1 SPKI Theory

SPKI certificates bind either names or explicit authorizations to keys or to other objects. Binding a key to another key can be either directly or indirectly through a name for the key or a hash of the key. Binding to an object (such as an executable file) is through a hash of the object or a name that resolves to that hash.

The standard format for SPKI certificates is the S-expression. SPKI use a simplified form of S−expressions; the canonical form. An S−expression is a list enclosed in "(" and ")". SEXP [88] is the S-expression technology used with added restriction that each list has a byte string as its first element. That first element is the "type" or "name" of the object represented.

SPKI objects are defined using an extension of BNF in which "|" means logical OR, "*" means closure (0 or more occurrences), "?" means optional (0 or 1 occurrence) and "+" means non-empty closure (1 or more occurrences). A quoted string represents a string literal.

[1] Simple Public Key Certificate
[2] Simple Distributed Security Infrastructure

7.1.1 Primitive objects

Prover to verifier communication involves a small number of different objects: a <sequence> of <cert>, <pub-key>, <signature> and <op> and the verifier's own <acl>. These are considered the top level objects. Before describing these top level objects we describe some primitive objects.

1. <pub-key>
   `<pub-key>:: "(" "public-key" "(" <pub-sig-alg-id> <s-expr>* ")"`
   `<uris>? ")" ;`
   A public key definition gives information required to employ the key for checking signatures. There are two pub-sig-alg-ids for signature verification defined so far; one for RSA key format and the other for DSA key format.
2. <hash>
   `<hash>:: "(" "hash" <hash-alg-name> <hash-value> <uris>? ")" ;`
   A <hash> object gives the hash value of some object.
3. <signature>
   `<signature>:: "(" "signature" <hash> <principal> <sig-val> ")" ;`
   `<sig-val>:: "(" <pub-sig-alg-id> <sig-params> ")" ;`
   `<sig-params>::<byte-string> | <s-expr>+ ;`

The <signature> value typically follows a <cert> object in a <sequence>. It is used for a certificate body. <sig-params> depends on the <pub-sig-alg-id>, the verification algorithm of the public key being used to verify this signature.

7.1.2 Authorization Certificates

```
 <cert>:: "(" "cert" <version>? <cert-display>? <issuer>
<issuer-loc>? <subject> <subject-loc>? <deleg>? <tag> <valid>?
<comment>? ")" ;
```

An authorization certificate transfers some specific authorization or permission from one principal to another. These certificates merely transfer authorization that is created by an ACL-entry. An ACL entry resides on the verifier's machine. Authorization flow is therefore a chain from the verifier's ACL through authorization certificates and then back to the verifier.

1. <issuer>

```
<issuer>:: "(" "issuer" <principal> ")" ;

<principal>:: <pub-key>|<hash-of-key> ;
```

<hash-of-key> is the preferred <principal> due to size and as a way of protecting small keys from cryptanalysis by keeping them secret.

2. <subject>

```
<subject>:: "(" "subject" <subj-obj> ")" ;\\
<subj-obj>:: <principal> | <name> | <obj-hash> | <keyholder> | <subj-thresh> ;\\
```

The subjects of interest to us are <principal>, <name> (representing a group of principals), <obj-hash> and <subj-thresh>.

`<subj-obj>:: <principal> ;`

is the most basic form of subject.

```
<obj-hash>:: "(" "object-hash" <hash> ")" ;
```

<obj-hash> option refers to an object rather than a <principal>. It is used to assign attributes to an object (a file, a web page, an executable program). An example would be assigning virus-free attribute to an executable file.

```
<subj-thresh>:: "(" "k-of-n" <k-val> <n-val> <subj-obj>* ")" ;
where K < N, and there are N <subj-obj> subjects listed.
```

A threshold subject specifies N subjects for a certificate or ACL entry, of which K must agree before the permission is delegated. The actual intention is to ensure that there are K distinct paths passing permission between the verifier's ACL and the prover's request.

At least K of the N subjects must show certificate paths which converge on a single target subject during reduction for the permission to be transmitted to the target. If there are fewer than K such paths then the permission is not delegated or exercised.

3. <deleg>

```
<deleg>:: "(" "propagate" ")" ;
```

This optional field indicates that the <subject> has the permission to delegate the permission given in the <tag> field fully or partially to others.

4. <tag>

```
<tag>:: "(" "tag" "(*)" ")" | "(" "tag" <tag-expr>  ")" ;
```

"(tag (*))" means "all permissions". The simplest tag is an S-expression with no *-forms. Such a tag defines specific permission which must be passed along and used intact. A tag with *-forms represents a set of specific permissions. Any subset of such a set of permissions may be delegated. To reduce 5-tuples from such certificates adjacent tag sets are intersected to find a resulting tag set.

All tags are assumed to be positional and extendable which means that parameters in a tag have a meaning defined by their position and every field added to the end of the permission restricts the permission granted.

Refer to appendix D for a complete BNF definition of the <tag> body. <tag> objects are defined by users in the language for describing sets of permissions given in the BNF. This is so that users can choose appropriate object names.

5. <valid>

```
<valid>:: "(" "valid" <not-before>? <not-after>? <online-test>*
")" ;
<not-after>:: "(" "not-after" <date> ")" ;
<not-before>::"(""not-before" <date> ")" ;
```

The <valid> field gives validity dates and/or online test information for the certificate. Either <not-after> or <not-before> or both dates may be omitted to make the certificate infinitely valid in either direction or both in directions. A date field is an ASCII byte string of the form: YYYY-MM-DD_HH:MM:SS and is always UTC.

```
<online-test>:: "(" "online" <online-type> <uris> <principal>
         <online-id> <s-part>* ")" | "(" "online" "new-cert" <uris> ")" ;
<online-type>:: "crl" | "reval" | "one-time" ;
<online-id>:: "(""id" <byte-string> ")" ;
```

The online test option allows finer grain validity testing. When online test specification is present in a certificate the certificate has to be validated from an online validity testing service (in our case the revocation service). The reply from the revocation service is a digitally signed object, validated by the <principal> given in the test specification. That reply object has validity dates, so that once one has the online test response, its validity dates can be intersected with the parent certificate's validity dates to yield the current working validity dates for the certificate.

A reply object can be of type crl, reval or one-time. We are interested in the re-validate (reval) type of response object that tells the verifier a list of valid certificates or just that the current certificate is valid.

The new-cert form requests a new copy of the certificate in question. We need this for certificates that are short-lived. A new-cert fetch is triggered by an expired certificate.

If the URI specifies an HTTP connection to the online test server, the URI can be used to provide all parameters needed (such as a hash of the certificate in question), but in other cases, such parameters might need to be listed in the optional <s-part>. The protocol used for these online tests is not fully specified here.

7.1.3 Name Certificates

This section gives the form of a name, a name certificate and the rules for reducing name certificates to simple mappings from name to key for trust computations. An issuer needs no authorization to create names and there is no "certification practice statement" for these name certificates. Nothing is implied by a name certificate about the principal(s) being named. A name is an arbitrary string assigned by the issuer (in issuer's name space) and is meaningful only to that issuer, although other parties may end up using it.

1. Name certificates syntax

```
(cert
    (issuer (name <principal> <name>))
    <subject>
    <valid>
)
<name-cert>:: "(" "cert" <version>? <cert-display>? <issuer-name>
<subject> <valid>? <comment>? ")" ;
<issuer-name>:: "(" "issuer" "(" "name" <principal> <byte-string> ")" ")" ;
```

The name is under the <principal> name space where <principal> is certificate issuer. Under this syntax the certificate issuer <principal> is taken from the (name..) structure.

The (tag) field is omitted and (tag (*)) is assumed. There is also no <deleg> field. A name definition is like an extension cord, passing everything the name is granted through to the subject. The subject is unrestricted.

If there are more than one name certificates for a given name with different subjects then that name is a group. In this sense all name certificates define groups, many of which will have only one member. A multi-member group is like a multi-plug extension cord, passing everything the name is granted through to any and all of its subjects.

2. Name certificates syntax

<name> form is an option for <subject> when one wants to generate a certificate granting authorization to a named group of principals This can be either a relative name or a fully-qualified name.

```
<name>:: <relative-name> | <fq-name> ;
<relative-name>:: "(" "name" <names> ")" ;
<fq-name>:: "(" "name" <principal> <names> ")" ;
<names>:: <byte-string>+ ;
```

A relative name is defined with respect to an issuer and should only show up in a certificate borrowing the <principal> from the issuer of that certificate. For evaluation purposes relative names are translated into a fully-qualified names before reduction. Unlike <issuer-name> which has to be a name in the issuer's name space the subject name can be in any name space.

2. Name reduction

Below is an example of name reduction from [42]. Given a name definition such as

```
(cert
   (issuer (name (hash md5 |Txoz1GxK/uBvJbx3prIhEw==|) fred))
   (subject (hash md5 |Z5pxCD64YwgS1IY4Rh61oA==|))
   (not-after "2001-01-01_00:00:00"))
```

the name

```
(subject (name (hash md5 |Txoz1GxK/uBvJbx3prIhEw==|) fred sam george mary))
```

reduces to

```
(subject (name (hash md5 |Z5pxCD64YwgS1IY4Rh61oA==|) sam george mary))
```

This is repeated until the name reduces to a key.

7.1.4 ACL and Sequence formats

1. <acl>

```
<acl>:: "(" "acl" <version>? <acl-entry>* ")" ;
<acl-entry>:: "(""entry" <subj-obj> <deleg>? <tag> <valid>?
<comment>? ")" ;
```

An ACL is a list of assertions; certificates that don't need issuer fields or signatures because they are held in secure memory. Since the fields of the ACL are similar to a <cert> we do not repeat those common field definitions. Since ACLs are never communicated, developers are free to choose their own formats. We will define one for our grid policy implementation.

The subject is given the permission specified in <tag> and if no expiration date or condition is specified this holds until the ACL is edited to remove the permission.

2. <Sequence>

```
<sequence>:: "(" "sequence" <seq-ent>* ")" ; <seq-ent>:: <cert> |
<pub-key> | <signature> | <crl> | <delta-crl> |
<reval> | <op> ;
<op>:: <hash-op> | <general-op> ;
<hash-op>:: "(" "do" "hash" <hash-alg-name> ")" ;
<general-op>:: "(" "do" <byte-string> <s-part>* ")"
```

A <sequence> is an ordered sequence of objects that the verifier will consider for authorization. By reducing certificates in the sequence, the verifier will establish that the final subject (key or object) has been granted authority through the sequence.

<op> codes are instructions to the verifier. So far the only opcode defined is "hash" with the meaning that the previous item in the sequence is to be hashed by the given algorithm and saved indexed by that hash value. The item is presumably referred to by its hash in a subsequent object. At present <signature> does double duty by calling for the hash of the preceding item as well.

7.1.5 Online test reply format

An online test result is a digitally signed object carrying its own date range, explicitly or implicitly. The reply can be a crl , a revalidation or a one-time revalidation. We are going to use revalidation which specifies that a given certificate (or list of certificates) is still valid. These replies have a <valid-basic> time interval. A reply is only valid for that interval and a sequence of replies has to be issued for non-overlapping intervals. The interval has to be systematically selected, its not an arbitrary value.

```
<reval>:: "(" "reval" <version>? <reval-list> <valid-basic> ")" ;
<reval-list>:: "(" "valid" <hash>+ ")" ;
```

A certificate is valid if it is listed in <reval-list>. The <reval> must be signed by the principal indicated in the <online-test> field that directed it to be fetched.

7.1.6 5-Tuple Reduction

This section describes trust evaluation that is part of every verifier which accepts SPKI certificates. The inputs to that trust engine are 5-tuples and Access Control List (ACL) entries which can also be translated to 5-tuples. This describes reduction of 5-tuples with principals and no names.

1. <5-tuple> BNF

We assume a 5-tuple with a construct of the form.

```
<5-tuple>:: <issuer5> <subject5> <deleg5> <tag-body5> <valid5> ;
<issuer5>:: <key5> | "self" ; <subject5>:: <key5> | <obj-hash> |
<keyholder> | <threshold-subj> ;
<deleg5>:: "t" | "f" ;
<key5>:: <pub-key> ;
<valid5>:: <valid-basic> | "null" | "now" ;
<tag-body5>:: <tag-body> | "null" ;
```

"self" is provided for ACL entries. The only meaningful 5-tuples after reduction is done are those with "self" as issuer.

2. Top level reduction rule

<i1,s1,d1,a1,v1> + <i2,s2,d2,a2,v2> yields <i1,s2,d2,a,v> if s1 = i2, d1 = "t", a = the intersection of a1 and a2 and v = the intersection of v1 and v2

Validity intersection involves normal intersection of date ranges if there are <not-before> or <not-after> fields in v1 or v2 and union of online tests if these are present in v1 or v2. Each online test includes a validity period and therefore there is a resulting validity interval in terms of dates. The intersection of a1 and a2 is given below. In the most basic case,
If a1 is (tag (*)), a = a2.

If a2 is (tag (*)), a = a1.
If a1 == a2, a = a2.
Otherwise, a = "null" and the 5-tuple doesn't reduce.

3. Intersection of tag sets

Two <tag> S-expressions intersect by the following rules. Note that in most cases, one of the two tag S-expressions will be free of *-forms. A developer is free to implement general purpose code that does set-to-set reductions, for example, but that is not likely to be necessary.

1. basic: If a1 $\equiv$ a2, then the result is a1.
2. basic: if a1 $\neq$ a2 and neither has a *-form, then the result is "null".
3. (tag (*)): if a1 $\equiv$ (tag (*)), then the result is a2. If a2 $\equiv$ (tag (*)), then the result is a1.
4. (* set ...): if some <tag> S-expression contains a (* set) construct expand the set and intersect the resulting simpler S-expressions. <item (* range ...): If a <tag> field compares a (* range) to a <byte-string>, do the specified range comparison and the resulting field is the explicit one tested. If the strings being compared have unequal display types the result is the empty set.
5. (* prefix ...): If some <tag> field compares a (* prefix) to a <byte-string> the result is the explicit string if the test string is a prefix of it and otherwise "null".

4. Reduction of subject threshold certificates

To reduce a subject threshold certificate, make k copies of the 5-tuple containing the K-of-N subject and indicate which subject is being handled by each copy. Reduce each copy independently and stop the separate reductions when K of the reduced values have the same subject. At that point, the K reduced 5-tuples become a single 5-tuple.

5. Certificate Result Certificates

After reducing a chain of certificates to the form:

```
(Self,X,D,A,V)
```

verifiers with a private key can sign the reduced certificate using the private key to obtain an SPKI certificate. This certificate can then be used by the prover, instead of the full chain, when speaking to that particular verifier during its validity period.

7.2 Specifying Grid Security Policy in SPKI

7.2.1 Identifying principals

For a detailed description refer to sections 4.2.1, 5.1.1 and 6.6. If the principal has an identity as is the case with users and Kerberos principals then we use a name certificate for the identity.

```
(cert
     (issuer (name (hash sha1 |TLCgPLFlGTzgUbcaYLW8kGTEnUk=|) principalID))
     (subject (hash sha1 |Ve1L/7MqiJcj+LSa/l10fl3tuTQ=|))
     (tag xxx)
     (not-before "1998-03-01_12:42:17")
     (not-after "2012-01-01_00:00:00")
)
```

When the principal has no identity, we define a name certificate where the name is a hash of the key.

```
(cert
     (issuer (hash sha1 |TLCgPLFlGTzgUbcaYLW8kGTEnUk=|)
             (hash sha1 |Ve1L/7MqiJcj+LSa/l10fl3tuTQ=|) )
     (subject (hash sha1 |Ve1L/7MqiJcj+LSa/l10fl3tuTQ=|))
     (tag xxx)
     (not-before "1998-03-01_12:42:17")
     (not-after "2012-01-01_00:00:00")
)
```

Omitting the dates makes the keys infinitely valid.

7.2.2 Trusted Third Parties

For a detailed description refer to section 4.2.2.

```
(cert
     (issuer (hash sha1 |TLCgPLFlGTzgUbcaYLW8kGTEnUk=|))
     (subject (hash sha1 |Ve1L/7MqiJcj+LSa/l10fl3tuTQ=|))
     (propagate)
     (tag trusted|partially trusted|untrusted|unknown)
)
```

Issuer trusts subject and subject principal is allowed to propagate this trust to introduce new principals to issuer.

7.2.3 Roles and Groups

For a detailed description refer to sections 4.2.3 and 4.2.4. Below are steps followed to create a group and define group members. K-group-owner is the public key for the owner of the group and K-group is the public key for the group. k-group-member-i is the key for the i^{th} member of the group.

1. Bind a name to the group key.
   ```
   (cert
        (issuer (name (hash sha1 K-group-owner) group-name))
        (subject (hash sha1 K-group})
   )
   ```
   The group's name can also be a hash of the groups public key.
2. Authorize the group owner key to be able to speak for the group key so that group owner key is able to sign membership certificates.
   ```
   (cert
        (issuer (name K-group-owner group-name))
        (subject K-group)
        (propagate)
        (tag *)
   )
   ```
3. Define group members by binding group name to key.
   ```
   (cert
        (issuer (name k-group-owner group-name))
        (subject k-group-member-i)
   )
   ```

Rights can be delegated to the group key and these are passed to the group members automatically through the certificates above.

7.2.4 Roles and Programs

For a detailed description of how roles can be utilized to securely load software refer to sections 4.2.3 and 4.2.5. To load a program named emacs securely, the node needs a certificate issued by node CA that binds a hash of the executable emacs.exe to the program name emacs. Let k-ca be the public key of the node CA and P be the program name. The CA can issue the following certificate.

```
(cert
 (issuer k-ca)
 (subject
  (object-hash
   (hash md5 |szKSlSK+SNzIsHH3wjAsTQ==| emacs.exe)))
 (tag (name "emacs"))
)
```

Suppose emacs is a program written by a different user from CA and this user will continue to update emacs.exe. The proper thing would be for CA to delegate the right to certify emacs to this user so that the user can issue a certificate of the hash of new versions of emacs.exe. Let k-user be the public key for the user. The authorization certificate from CA to user would be:

```
(cert
    (issuer (name k-CA emacs-owner))
    (subject k-user)
    (tag (name "emacs"))
)
```

and the secure loading certificate from user for emacs.exe would be.

```
(cert
 (issuer k-user)
 (subject (object-hash (hash md5 |szKSlSK+SNzIsHH3wjAsTQ==| emacs.exe)))
 (tag (name "emacs"))
)
```

7.2.5 Protecting Nodes

For a detailed description on how to determine that software is trusted or not in order to protect nodes executing it refer to section 4.2.12. The node CA creates a group called trustedSW as described in section 7.2.3. To make a program digest speak for the group (i.e trusted), the following certificate will suffice:

```
(cert
 (issuer (name k-ca trustedSW) )
 (subject
  (object-hash
   (hash md5 |szKSlSK+SNzIsHH3wjAsTQ==| emacs.exe)))
)
```

Alternatively, the CA can trust a user and allow user to issue trust certificates to programs. CA has to authorize the user to make these kind of certificates. CA creates a group called trustedSW-*owner* and makes user U a member of this group. Members of this group are trusted and so are any programs that they trust.

1. ```
   (cert
    (issuer (name k-ca trustedSW-owner) )
    (subject k-user)
   )
   ```
   ```

2. (cert
    (issuer k-ca)
    (subject (name k-ca trustedSW-owner))
    (tag *)
)
3. (cert
    (issuer (name k-ca trustedSW))
    (subject
     (object-hash
      (hash md5 |szKSlSK+SNzIsHH3wjAsTQ==| emacs.exe)))
)

7.2.6 Delegation

For a detailed description refer to section 4.2.6.

7.2.6.1 Delegation Identity

```
(cert
     (issuer (hash sha1 |TLCgPLFlGTzgUbcaYLW8kGTEnUk=|))
     (subject (hash sha1 |Ve1L/7MqiJcj+LSa/l10fl3tuTQ=|))
     (tag (*))
     (not-before "1998-03-01_12:42:17")
     (not-after "2012-01-01_00:00:00")
)
```

tag (*) means all permissions and is therefore equivalent to delegating an identity.

7.2.6.2 Delegating rights

```
(cert
     (issuer (hash sha1 |TLCgPLFlGTzgUbcaYLW8kGTEnUk=|))
     (subject (hash sha1 |Ve1L/7MqiJcj+LSa/l10fl3tuTQ=|))
     (<tag>)
     (not-before "1998-03-01_12:42:17")
     (not-after "2012-01-01_00:00:00")
   )
```

<tag> indicates the permissions delegated by the certificate. Examples of <tag> permissions given by [41] are:

1. FTP

   (tag (ftp cybercash.com cme))
   This <tag> indicates that the Subject has permission to do FTP into host cybercash.com as user cme.
2. HTTP
   (tag (http http://acme.com/company-private/personnel))
   This <tag> gives the Subject permission to access the web page at the given URI. To give permission for an entire tree under a given URI, one might use:
   (tag (http (* prefix http://acme.com/company-private/personnel/
   )))

3. TELNET
   `(tag (telnet clark.net cme ))`
   This <tag> gives the Subject permission to telnet into host clark.net as user cme.

7.2.6.3 Joint Delegation

User to node delegations (login, section 5.1.3) is a case of joint delegation in which the user (key k_u) is delegating to a short term login key (K_l) and to a long term key (k_w). To achieve this in SPKI:

1. key k_u issues a 2-of-2 threshold certificate to K_l and k_w
   ```
   (cert
     (issuer k-u)
     (subject ("2-of-2") 2 2 (public-key k-l rsa-pkvs1-md5)
                             (public-key k-w rsa-pkvs1-md5))
     (tag (*))
   )
   ```
2. Both must delegate to a common subject in order for the delegation to be valid. The node key k_w is the one that needs this delegation. Assuming that by default every key speaks for itself it is enough for k_l to delegate a short term key (refreshed every 30 minutes) to k_w. Without this assumption then k_w would need to delegate to itself. Delegation from k_l to k_w is as follows:
   ```
   (cert
     (issuer k-l)
     (subject k-w)
     (tag (*))
     (not-before "1999-03-01_12:00:00")
     (not-after "1999-03-01_12:30:00")
   )
   ```
   If k_l is forgotten (typically after user logout) and above delegation certificate expires the node can no longer speak for the user. The objective has been achieved that user-to-node delegation ends after user logs out.

7.2.7 Revoking Delegations

For detailed description refer to sections 4.2.8, 5.10, 7.1.2 and 7.1.5. The idea is that a principal that is issuing a certificate that might be cancelled later has to specify additional information on the certificate so that the verifier is able to determine whether this certificate has been cancelled or not. The following is an example of a certificate that might be cancelled by a principal. It has a specification of information required by a verifier to check whether certificate is revoked or not.

```
(cert
  (issuer (hash sha1 |TLCgPLFlGTzgUbcaYLW8kGTEnUk=|))
  (subject (hash sha1 |Ve1L/7MqiJcj+LSa/l10fl3tuTQ=|))
  (tag (*))
  (online reval http.//xyz (hash sha1 |TLCgPLFlGTzgUbcaYLW8kGTEnUk=|)
   <online-id>)
)
```

The specified key should sign the revalidation list for this certificate.

7.2.8 Renewing Certificates

For detailed description refer to sections 4.2.8, 5.10, 7.1.2 and 7.1.5. Most of the certificates used are short term certificates that expire after a certain duration has passed. They therefore need to be constantly renewed. SPKI has a simplified way of specifying information required to renew such certificates.

```
(cert
  (issuer (hash sha1 |TLCgPLFlGTzgUbcaYLW8kGTEnUk=|))
  (subject (hash sha1 |Ve1L/7MqiJcj+LSa/l10fl3tuTQ=|))
  (tag (*))
  (online new-cert <uris>)
)
```

The URI is the location of the service that will issue a new certificate.

7.2.9 Authorization

For a detailed description refer to sections 4.2.10, 4.2.9, 4.4.1 and 4.4.2.

7.2.9.1 Permissions Management

ACL entries are represented as:

```
(acl
    (entry
     (name (hash md5 |p1isZirSN3CBscfNQSbiDA==|) sysadmin/operators)
     (tag (ftp db.acme.com root)))
)
```

or as a 5-tuple

```
 (self ,(public-key
(rsa-pkcs1-md5 <s-expr>*)), t, (ftp db.acme.com root), null)
```

<s-expr> is the public key of the subject.

If the acl demands that a principal must speak for several principals in the acl before granting access, such as have the right delegation and run from a trusted computer, then we use a 5-tuple in which the subject is a k-of-n. An example:

```
( self ,
  (2-of-2 2 2 (public-key (rsa-pkcs1-md5 <s-expr>*))
                  (name someTrustedNode)) ,
  t,
  (ftp db.acme.com root),
  null
)
```

This certificate means that there should be a chain of delegation anchored by the first principal and there should be a second chain anchored by the second principal. The second principal, someTrustedNode, indicates that the node possess a certificate from verifier's CA.

7.2.9.2 Authorization algorithm

Steps to be followed in authorization.

1. Reduce all names to keys using the algorithm in section 7.1.3; name reduction.
2. Convert all certificates and ACLs to 5-tuples.
3. Perform 5-tuple reduction using the algorithm in section 7.1.6; top level reduction. Rules for intersection of rights are as stated in section 7.1.6; intersection of tag sets. Threshold certificates are reduce using the algorithm in section 7.1.6; reduction of threshold certificates.

7.3 Implementation Proposal

Figure 7.1 shows what the system components are in an implementation. The objective of dividing the system into the components identified is so that it is possible to adopt existing implementation of some of the components. In particular we would like to look for existing implementations of a PKI for Simple Public Key Certificates (SPKI).

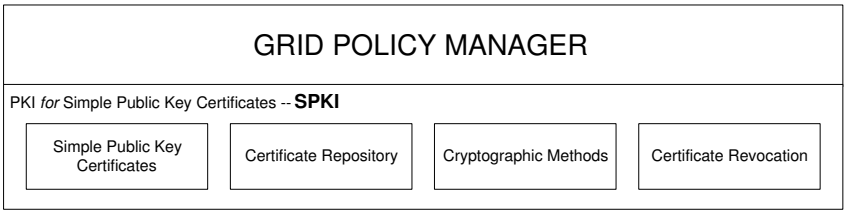

Figure 7.1: Components of a Grid Security System

To understand what we will be looking for a definition of PKI might help. From [99], a PKI is a framework of people, processes, policies, protocols, hardware, and software used to generate, manage, store, deploy and revoke public key certificates. In our case the certificates are SPKI certificates rather than X.504 v3 certificates as is the case in [99]. In the context of X.509 v3 certificates, all the possible logical components of PKI are:

- End entities or subscribers
- Certificate authorities
- Certificate policies
- Certificate practices statement
- Hardware security modules
- Public key certificates
- Certificate extensions
- Registration authorities
- Certificate depositories

We won't go into further description of what PKI is, we will in the following sections describe SPKI specifics relevant to our proposed implementation.

7.3.1 Grid Policy Manager

The policy manager is the implementation of the grid policy. The manager is not necessarily one huge monolithic component. It could be broken down further into sub-components but we don't get into that.

7.3.2 SPKI - Simple Public Key Certificates

7.3.2.1 JDK 1.2 Cryptography Architecture

Partanen et al. [83] has made an extension to JDK 1.2 Java Cryptography Architecture (JCA) to add SPKI certificates. JCA is based on the design principles of independence, interoperability and extensibility. The aim is to have users of the API use cryptographic concepts without thinking about their implementation and neither the algorithms used for the implementation.

Algorithm independence is achieved by defining types of cryptographic services called "engines" and defining classes that provide the functionality of these engines. Implementation independence is achieved using a "provider-based" architecture. The term provider refers to a package or set of packages that implement one or more cryptographic services. A request for a cryptographic service should specify the provider or else get the default provider.

The architecture has therefore got several levels. The top level consist of interfaces and abstract classes for example *certificate*. The second level consists of abstract classes and interfaces for different subtypes of the general concepts. As an example the *certificate* class is extended by *X509Certificate* class. The actual implementation is via one or more providers. The default provider "SUN" comes with JDK and implements most of the concepts.

7.3.2.2 Extending the Certificate class

The *java.security.cert.Certificate* class is an abstract class for certificates. The default subclass to this class is the X509Certificate class. It is an abstract class serving all X509 certificate implementations.

JDK provides certificate and key storage functionality as part of key management. In JDK the *keystore* class represents storage facility that can contain keys and certificates. The actual implementation can be chosen from one of available implementations. JDK includes one available implementation, *java-Keystore* class that stores the objects in an encrypted file using a proprietary file format. Providing an implementation that stores objects differently is possible. In our grid security system we need an implementation that can store these objects in the DNS.

In this proposal the *Certificate* class is extended with a new abstract subclass called *SPKICertificate* for SPKI certificates.

7.3.2.3 Implementing SPKI Certificates

The core of the implementation are the *SPKICertificate* and *SPKICert* classes. The *SPKICertificate* is the abstract super class for all SPKI certificates and the *SPKICert* class is the actual implementation. Figures 7.2 shows the complete class hierarchy for SPKI certificates. The specification is in UML (Unified Modelling Language).

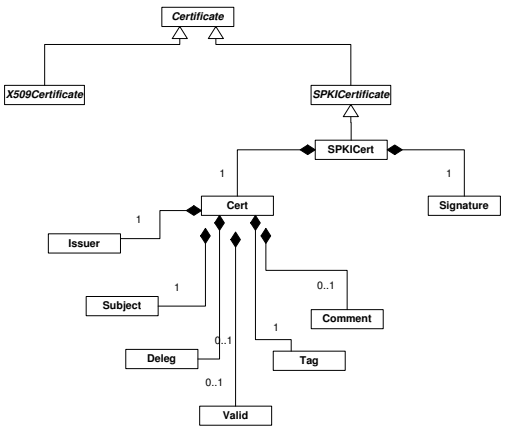

Figure 7.2: SPKI certificate object structure

As explained above, Java uses a class called *Provider* to find the classes implementing particular services. To register SPKI functionality, a provider is required. This provider is called *SPKIProvider*. It extends *java.security.provider* class and specifies the names of classes providing functionality for handling SPKI certificates which in this case is the *SPKICertificateFactory* class.

The *SPKICertificateFactory* is an engine class that is used to create *SPKICertificate* objects from canonical s-expressions. It extends the *CertificateFactory* class. A utility class *SPKIParserVisitor* is used in parsing canonical formats into SPKI object structures. The provider class should also include the *KeyStore* implementation.

7.3.3 SPKI - Certificate Repository

7.3.3.1 Overview

Nikander et al. [80] has proposed a way to store and retrieve SPKI certificates using DNS. The proposal has been implemented into a prototype.

The main idea behind storing certificates in the DNS is to have DNS nodes that carry resource records for a specific SPKI key (principal). The domain name of this node can be given as the optional issuer-location and/or subject-location fields of each certificate.

The decision whether to store a given certificate at its issuer DNS node, subject DNS node or both depends on the certificate type. In our case, name certificates are to be stored at the subject node since they are attributes of the subject and authorization certificates are stored at both the issuer and subject node.

7.3.3.2 Search algorithm

The proposed search algorithm is an extension to the search algorithm presented by Aura [93]. The algorithm adds a number of heuristics to reduce the cost of searching.

Stored certificates form directed delegation networks in which issuers and subjects are nodes while the certificates are represented by the arcs. An arc connects an issuer to a subject which is an abstract representation of a certificate. The search problem is to find a path from the verifier to the final subject thereby creating a chain. The permission being checked should be transferred on all arcs in the chain and all restrictions that can potentially break the chain should considered.

7.3.3.3 Certificate Administration

From an administrator's point of view certificates are created, stored in the DNS for retrieval, and removed from the DNS once they expire or have been revoked. The task of adding and removing certificates is not different from any other DNS administration task.

7.3.4 SPKI - Cryptographic Methods

Functions that should be included into the cryptographic methods component are:

1. Digital Signatures,
2. Hash Functions,
3. Public Key Cryptosystems,
4. Secret Key Cryptosystems.

7.3.4.1 TCB implementation of cryptographic methods

Due to the fact that cryptographic methods are well understood, well established in terms of the theory and fairly standard it is our proposal that the libraries implementing these functions should be part of the OS running the grid nodes. Support for cryptographic methods would in this case be one of the requirements of the OS software used in the grid.

7.3.4.2 JAVA implementation of cryptographic methods

Java^{TM} 2 SDK, Standard Edition, v 1.4 has a set of packages called Java^{TM} Cryptography Extension (JCE), described in [71], that provide a framework and implementations for encryption, key generation

and key agreement and Message Authentication Code (MAC) algorithms. Support for encryption provided includes symmetric, asymmetric, block and stream ciphers, secure streams and sealed objects. The JCE API covers:

- Symmetric bulk encryption, such as DES, RC2 and IDEA,
- Symmetric stream encryption, such as RC4,
- Asymmetric encryption, such as RSA,
- Password-based encryption (PBE),
- Key agreement,
- Message Authentication Codes (MAC).

The Java 2 SDK v 1.4 comes standard with a JCE provider named "SunJCE" which comes pre-installed and registered. It provides the following cryptographic services:

- An implementation of DES, triple DES, and Blowfish encryption algorithms in Electronic Code Book (ECB), Cipher Block Chaining (CBC), Cipher Feedback (CFB), Output Feedback (OFB) and Propagating Cipher Block Chaining (PCBC) modes.
- Key generators for generating keys for DES, triple DES, Blowfish, HMAC-MD5, and HMAC-SHA1 algorithms.
- An implementation of the MD5 with DES-CBC password-based encryption (PBE) algorithm defined in PKCS ♯5.
- "Secret-key factories" providing bi-directional conversions between opaque DES, Triple DES and PBE key objects and transparent representations of their underlying key material.
- An implementation of the Diffie-Hellman key agreement algorithm between two or more parties.
- A Diffie-Hellman key pair generator for generating a pair of public and private values suitable for the Diffie-Hellman algorithm.
- A Diffie-Hellman algorithm parameter generator.
- A Diffie-Hellman "key factory" providing bi-directional conversions between opaque Diffie-Hellman key objects and transparent representations of their underlying key material.
- Algorithm parameter managers for Diffie-Hellman, DES, Triple DES, Blowfish, and PBE parameters.
- An implementation of the HMAC-MD5 and HMAC-SHA1 keyed-hashing algorithms defined in RFC 2104.
- An implementation of the padding scheme described in PKCS ♯5.
- A keystore implementation for the proprietary keystore type named "JCEKS".

7.3.5 SPKI - Certificate Revocation

An implementation has to be made according to the description in sections 4.2.8 and 5.10. There are no existing implementations found.

Chapter 8

Grid Policy Management using Trust-Management Engines

This chapter will investigate the use of trust-management engines for enforcing the grid security policy specified in chapter 4. We explore the use of these systems along the same lines as has been done for SPKI in chapter 7. This should give us some kind of sufficient comparison between SPKI and these trust-management engines to be able to decide which of the two would be more suitable for implementation.

8.1 Trust Management Approach

SPKI, already described in chapter 7, is one of two trust management approach mechanisms developed as an answer to inadequate identity-oriented approach mechanisms. The second trust management approach is the trust-management engines. There are two existing trust-management engines: PolicyMaker and Keynote.

In the trust management approach to security the focus is on dealing with the trust-management problem rather than the security needs of one particular service. Aspects of the trust-management problem include formulating security policies and security credentials, determining particular sets of credentials satisfy relevant policies and deferring trust to third parties.

Trust-management, Blaze et al. [20], provides a unified approach to specifying and interpreting security policies, credentials and relationships. The notion of specifying security policy is combined with the mechanism for specifying security credentials.

The *trust-management approach* frames the question of authorization as follows: "Does the set C of *credentials* prove that the *request* r *complies* with the local security *policy* P?"

Trust management systems provide a unified mechanism for specifying security polices and credential. The credentials describe specific delegation of trust and assume the role of public key certificates that directly bind keys to authorization. A trust-management system has five basic components:

- A language for specifying 'actions'. These are the operations that are controlled by the system.
- A mechanism for identifying 'principals', which are entities that can be authorized to perform actions.
- A language for specifying application 'policies', which govern the actions that principals are authorized to perform. Trust management policies are easy to distribute across networks which helps to avoid the need for application specific distributed policy configuration mechanisms, access control lists, and certificate parsers and interpreters.
- A language for specifying 'credentials', which allow principals to delegate authorization to other principals.

- A 'compliance checker', which provides a service to applications for determining how an action requested by principals should be handled, given a policy and a set of credentials
- Actions are described to the Keynote compliance checker using a collection of name-value pairs called an 'action attribute set'. This set is created by the calling application.

8.2 Trust-management Engines

The *trust-management engine* is a system component that takes (r,C,P) as input and returns a decision whether the request *complies* with the local policy or not. Section 8.1 defines r as the request, C as the set of credentials and P as the security policy. This definition of a trust-management engine is depicted in figure 8.1.

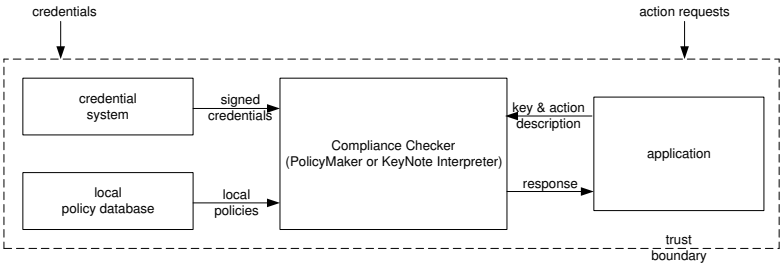

Figure 8.1: Trust Management Engine Architecture [16]

SPKI is a trust-management approach but it is not a trust-management engine because compliance checking may be done in an application-dependent manner. If the processing plan presented in [42] was universally adopted, SPKI would be a trust-management engine.

8.2.1 PolicyMaker

PolícyMaker was the first "trust-management engine". It addressed the authorization problem directly and provided an application-independent definition of "proof of compliance" for matching requests, credentials and policies. A full description of PolicyMaker is found in [20] and its compliance checking algorithm in [21].

PolicyMaker credentials and policies are fully programmable. Credentials and policies are together referred to as "assertions". In a policy assertion, the source is the keyword **POLICY** and in credential assertion the source is the public key of the issuing authority. Credentials must be signed by their issuers.

PolicyMaker can be written in any programming language that can "safely" be interpreted by a local environment. A safe version of AWK was developed for experimental work on PolicyMaker. A credential issued by a particular authority is useful in supporting a request only if the recipient of the request has an interpreter for the language in which the assertion is written. The "proof of compliance" and "assertion language design" are orthogonal and can be worked independently. At least one of the assertions presented for an evaluation must be a policy assertion that acts as the "root trust".

To make PolicyMaker minimal and analyzable as much responsibility was placed on the calling application as possible. The calling application is for instance responsible for all cryptographic verification of signatures on credentials. An additional crucial responsibility for applications is gathering credentials necessary for a request.

The main technical contribution of the PolicyMaker project is the notion of "proof of compliance" that is fully specified and analyzed. The most general version of the compliance checking problem allows assertions to be arbitrary functions. However, the computationally tractable version that is analyzed

and implemented requires all assertions to be monotonic. In particular policy assertions that make use of "negative credentials" such as revocation lists are excluded. The option is to convert non-monotonic policies to monotonic policies. For example rather than require that an assertion does not appear in a revocation list it should be required that the assertion should present a "certificate of non-revocation". This way no dangerous action can be performed due to absence of negative credentials.

8.2.2 Keynote

Keynote [18] was designed with the same principles as PolicyMaker with additional goals of standardization and ease of integration into applications. To meet these additional goals, Keynote assigns more responsibility to the trust-management engine than PolicyMaker and less to the calling application. Cryptographic signature verification is for instance done by the trust-management engine in Keynote. KeyNote requires that credentials and policies be written in a specific assertion language.

A calling application passes to a KeyNote evaluator a list of credentials, policies, requestor public keys and an "Action Environment". This last element is list of attribute/value pairs similar in some way to the Unix^{TM} shell environment. The action environment is constructed by the calling procedure and contains all information relevant to a request and necessary for making a decision. The action-environment attributes and the assignment of their values must reflect the security requirements of the application. Identifying the attributes to be included in the action environment in the most important task of integrating KeyNote into new applications.

The KeyNote assertion format is similar to that of email headers. The following is a sample KeyNote assertion.

```
keyNote-Version: 1
Authorizer: rsa-pkcs1-hex:"1023abcd"
Licensees: dsa-hex:"986412a02" ||
           rsa-pkcs1-hex"19abcds03"
Comment: Authorizer delegates read access
         to either of Licensee
Conditions: ($file == "/etc/passwd" &&
             $access == "read")-> {return "ok"}
Signature: rsa-md5-pkcs1-hex."f00f5673"
```

Policies and credentials have the same format. Policies, unlike credentials, are locally trusted and do not need signatures. The *Licensees* field specify the principal or principals to which authority is delegated. In syntax the licensees field is a formula in which arguments are public keys and operators are conjunctions, disjunctions and threshold. The programs in KeyNote are encoded in the *Conditions* field and are essentially tests of the action environment variables.

KeyNote uses a depth-first search (DFS) algorithm that attempts to satisfy at least one policy assertion. Satisfying an assertion entails satisfying both the *Conditions* field and *Licensees* key expression. KeyNote's policy model is a subset of PolicyMaker's. For a request to be approved, an assertion graph must be constructed between one or more policy assertions and one or more keys that signed the request.

PolicyMaker's restriction regarding "negative credentials" also apply to KeyNote. CRLs are not built into KeyNote but they could be provided at a higher level transparently to KeyNote. The problem of credential discovery is also not explicitly addressed in KeyNote. These two are areas of further research [19].

8.2.3 KeyNote versus PolicyMaker

The basic design of KeyNote and PolicyMaker is the same but KeyNote's features have been simplified to more directly support public key infrastructure-like applications. The key differences are [22]:

- KeyNote predicates are written in a standard notation based on C-like expressions and regular expressions.
- KeyNote assertions always return a boolean answer.
- Credential signature verification is built into KeyNote.
- Assertion syntax is based on human-readable "RFC-822" style syntax.
- Trusted actions are described using simple attribute/value pairs.

KeyNote is more standardized and easier to integrate into applications. There has therefore been more activity with Keynote integration into applications than PolicyMaker. Examples of new applications of KeyNote are IPSEC security [24] and electronic payments [23]. The fact that KeyNote has also got an internet standard on the way [17] indicates that KeyNote is the way to go. PolicyMaker will primarily remain as a proposal for extending KeyNote whenever more complexity and flexibility is needed.

8.3 Keynote Trust Management System

8.3.1 KeyNote concepts

The following are definition of key terms used in KeyNote [14]:

- An *application* is any program or system that uses KeyNote to control its security policy.
- An *action* is any operation with security consequences that is to be controlled by the KeyNote system.
- A *principal* is an entity that can be authorized to perform actions. Principals can be identified by arbitrary names or by their public keys.
- A *request* is what a principal does when it wants an application to perform an action. The principal that makes a request is called the *action requester* or, alternatively, the *action authorizer*.
- A *policy* is a set of rules that governs the actions that principals are authorized to perform in some application.
- A *credential* is a signed message that allows a principal to delegate part of its own authority to perform actions to other principals. A public key "certificate" is an example of a credential in KeyNote terms.
- A *Policy Compliance Value* (PCV) is a result returned by the KeyNote system to the application. It describes whether an action requested by a principal conforms to an application's local policy.

The application receives a principal's request and then calls KeyNote with a correct description of the action, the policy, and the relevant credentials. The PCV returned by KeyNote then advises the application how to handle the request. Below is sample pseudocode showing how an application can use KeyNote [14]. This code is placed at each point when the application receives a request from a principal.

```
requester            = requesting principal's identifier;
action_description   = data structure describing action;
policy               = data structure describing local policy,
                             typically read from a local file;
credentials          = data structure with any relevant
                              credentials, typically sent along with
                              the request by the requesting principal;
PCV                  = Call_KeyNote(requester,
                                    action_description,
                                    policy,
                                    credentials);
If (PCV == "allowed")
   perform requested action
Else
   reject requested action
EndIf
```

8.3.2 KeyNote trust-management architecture

KeyNote offers a simple unified mechanism for specifying security policies and security credentials. Distributed applications can easily create policies that defer to a remotely-managed authority, which can change the policy simply by issuing new credentials (certificates). A policy can become a remotely-usable credential by being signed. Figure 8.2 shows KeyNote trust-management architecture.

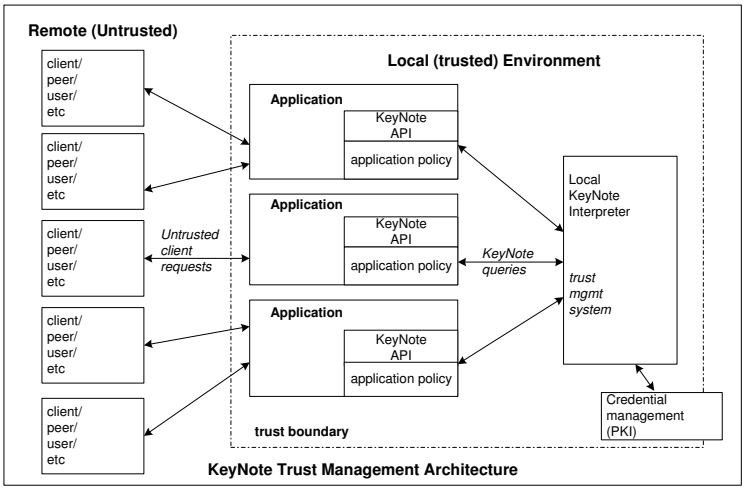

Figure 8.2: Trust Management Engine Application Architecture [14]

There is a clear separation of responsibilities between KeyNote, the application and the support infrastructure. The KeyNote compliance checker will determine, based on policy and credential assertion, whether a proposed action is permitted according to a policy. The use of KeyNote as a policy enforcement mechanism depends on a number of factors.

- Action attributes and their values must accurately reflect the security requirement of the application. Identifying the attributes to include in the action attribute set is the most important task in integrating KeyNote into new applications.
- The application policy must be sound and well formed.
- The application itself need to be trustworthy. KeyNote only provides advice to application and does not enforce policy directly. An application can disregard this advice or even submit false credentials.

It is the responsibility of the application to select appropriate credentials and policy assertions to run a query. Since KeyNote credentials are monotonic, omitting credentials can only result in legal actions being disallowed rather than illegal actions being allowed.

KeyNote does not provide credential revocation services, although credentials can be written to expire by some date. Applications that require revocation services can use KeyNote to specify and implement revocation policies.

There are several possible ways of implementing interfaces to KeyNote. The simplest implementation would be KeyNote running as a separate application that is called by applications that need service. Second option would be to have KeyNote implemented as a library that is linked into applications. Finally KeyNote can run a trusted service that other applications communicate to using interprocess communications mechanisms.

8.3.3 KeyNote assertions

8.3.3.1 Basic Structure

KeyNote assertions are described using a notation in which [X]* means zero or more repetitions of character string X, [X]+ means one or more repetitions of X, <X>* means zero or more repetitions of non-terminal <X>, <X>+ means one or more repetitions of X, and <X>? means zero or one repetitions of X. Nonterminal grammar symbols are enclosed in angle brackets while quoted strings in grammar productions represent terminals.

The basic structure of KeyNote assertions is as follows:

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

Action attributes provide the primary mechanism for applications to pass information to assertions. Attribute names are strings from a limited character set and attribute values are represented internally as strings.

Principals are represented as ASCII strings called 'Principal Identifiers'. Principal Identifiers may be arbitrary labels whose structure is not known to KeyNote or they may be cryptographic keys that are used by KeyNote for credential signature verification.

```
<PrincipalIdentifier>:: <OpaqueID>
                      | <KeyID> ;
<OpaqueID>:: <StrEx> ;
<KeyID>:: <IDString> ;
<IDString>:: <ALGORITHM>":"<ENCODEDBITS> ;
```

Principal Identifiers that are used by KeyNote only as labels are said to be 'opaque'. Opaque identifiers are encoded in assertions as strings. Unlike Opaque Identifiers, however, Cryptographic Identifier strings have a special form. "ALGORITHM" is an ASCII substring that describes the algorithms to be used in interpreting the key's bits. The ALGORITHM identifies the major cryptographic algorithm. "ENCODEDBITS" is a substring of characters representing the key's bits, the encoding and format of which depends on the ALGORITHM. Cryptographic Principal Identifiers are converted to a normalized canonical form for the purposes of any internal comparisons between them

8.3.3.2 The KeyNote-Version Field

This field identifies the version of the KeyNote assertion language in which the assertion was written. It has the form:

```
<VersionField>:: "KeyNote-Version:" <VersionString> ;
<VersionString>:: <StringLiteral>
                | <IntegerLiteral> ;
```

8.3.3.3 The Local-Constants Field

This field allows the use of short names rather than the usually lengthy cryptographic principal identifiers.

```
<LocalConstantsField>:: "Local-Constants:" <Assignments> ;
<Assignments>:: /* can be empty */
              | <AttributeID> "=" <StringLiteral> <Assignments> ;
```

<AttributeID> is an attribute name from the action attribute name space.

8.3.3.4 The Authorizer Field

The Authorizer field identifies the principal issuing the assertion. This field is of the form:

```
<AuthField>:: "Authorizer:" <AuthID> ;
<AuthID>:: <PrincipalIdentifier>
         | <DerefAttribute> ;
```

8.3.3.5 The Licensees Field

The Licensees field identifies the principals authorized by the assertion. Authorization can be distributed across several principals through the use of 'and', 'or' and threshold constructs. This field is of the form

```
<LicenseesField>:: "Licensees:" <LicenseesExpr> ;
<LicenseesExpr>::      /* can be empty */
                  | <PrincExpr> ;
<PrincExpr>:: "(" <PrincExpr> ")"
              | <PrincExpr> "&&" <PrincExpr>
              | <PrincExpr> "||" <PrincExpr>
              | <K>"-of(" <PrincList> ")"        /* Threshold */
              | <PrincipalIdentifier>
              | <DerefAttribute> ;
<PrincList>:: <PrincipalIdentifier>
            | <DerefAttribute>
            | <PrincList> "," <PrincList> ;
<K>:: {Decimal number starting with a digit from 1 to 9} ;
The ``&&'' operator has higher precedence than ``||'' operator.
<K> is an ASCII-encoded positive decimal integer.
```

8.3.3.6 The Conditions Field

This field gives the conditions under which the Authorizer trusts the Licensees to perform an action. "Conditions" are predicates that operate on the action attribute set. The conditions field is of the form.

```
<ConditionsField>:: "Conditions:" <ConditionsProgram> ;
<ConditionsProgram>:: /* Can be empty */
                    | <Clause> ";" <ConditionsProgram> ;
<Clause>:: <Test> "->" "{" <ConditionsProgram> "}"
         | <Test> "->" <Value>
         | <Test> ;
<Value>:: <StrEx> ;
<Test>:: <RelExpr> ;
<RelExpr>:: "(" <RelExpr> ")"        /* Parentheses */
          | <RelExpr> "&&" <RelExpr> /* Logical AND */
          | <RelExpr> "||" <RelExpr> /* Logical OR */
          | "!" <RelExpr>         /* Logical NOT */
          | <IntRelExpr>
          | <FloatRelExpr>
          | <StringRelExpr>
          | "true"        /* case insensitive */
          | "false" ;     /* case insensitive */
<IntRelExpr>:: <IntEx> "==" <IntEx>
             | <IntEx> "!=" <IntEx>
             | <IntEx> "<" <IntEx>
```

```
                | <IntEx> ">" <IntEx>
                | <IntEx> "<=" <IntEx>
                | <IntEx> ">=" <IntEx> ;
    <FloatRelExpr>:: <FloatEx> "<" <FloatEx>
                   | <FloatEx> ">" <FloatEx>
                   | <FloatEx> "<=" <FloatEx>
                   | <FloatEx> ">=" <FloatEx> ;
    <StringRelExpr>:: <StrEx> "==" <StrEx>  /* String equality */
                    | <StrEx> "!=" <StrEx>  /* String inequality */
                    | <StrEx> "<" <StrEx>   /* Alphanum. comparisons */
                    | <StrEx> ">" <StrEx>
                    | <StrEx> "<=" <StrEx>
                    | <StrEx> ">=" <StrEx>
                    | <StrEx> "~=" <RegExpr> ; /* Reg. expr. matching */
    <IntEx>:: <IntEx> "+" <IntEx>        /* Integer */
            | <IntEx> "-" <IntEx>
            | <IntEx> "*" <IntEx>
            | <IntEx> "/" <IntEx>
            | <IntEx> "%" <IntEx>
            | <IntEx> "^" <IntEx>        /* Exponentiation */
            | "-" <IntEx>
            | "(" <IntEx> ")"
            | <IntegerLiteral>
            | "@" <StrEx> ;
    <FloatEx>:: <FloatEx> "+" <FloatEx>  /* Floating point */
              | <FloatEx> "-" <FloatEx>
              | <FloatEx> "*" <FloatEx>
              | <FloatEx> "/" <FloatEx>
              | <FloatEx> "^" <FloatEx> /* Exponentiation */
              | "-" <FloatEx>
              | "(" <FloatEx> ")"
              | <FloatLiteral>
              | "&" <StrEx> ;
    <IntegerLiteral>:: {Decimal number of at least one digit} ;
    <FloatLiteral>:: <IntegerLiteral>"."<IntegerLiteral> ;
```

<StringLiteral> is a quoted string, <AttributeID> is as defined before and <RegExp> is a standard regular expression conforming to the POSIX 1003.2 regular expression syntax and semantics.

8.3.3.7 The Comment Field

The Comment field is used to annotate assertions with information describing their purpose. It is of the form:

```
<CommentField>:: "Comment:" <text> ;
```

The contents of this field are not interpreted. Comments can also be inserted anywhere into the assertion using the ♯ character.

8.3.3.8 The Signature Field

The signature field identifies a signed assertion and gives the encoded digital signature of the principal identified in the Authorizer field. The signature field is of the form.

```
<SignatureField>:: "Signature:" <Signature> ;
<Signature>:: <StrEx> ;
```

The <Signature> string should be of the form:

```
<IDString>:: <ALGORITHM>":"<ENCODEDBITS> ;
```

The formats of the "ALGORITHM" and "ENCODEDBITS" substrings are as described for cryptographic principal identifiers earlier. The algorithm name should be the same as that of the principal appearing in the Authorizer field. The IANA will provide a registry of reserved names. The encodings of the signature and the authorizer key do not have to be the same.

If the signature field is included, the principal named in the Authorizer field must be a cryptographic principal identifier. The algorithm must be known to KeyNote and the signature must be correct for the assertion body and authorizer key.

8.3.3.9 Sample KeyNote assertions

There are a number of sample assertions in appendix E from [17].

8.3.4 Shortcomings of KeyNote

8.3.4.1 Certificate revocation

There is no proposal for certificate revocation services. CRLs which are a common implementation of revocation are not applicable because they are not monotonic.

8.3.4.2 Certificates discovery

There is no proposal on how to gather or select the credentials required to authorize a request.

8.3.4.3 Certificate storage

There is no proposal on how to store credentials so they can be easily fetched for use in authorization.

8.4 Specifying Grid Security Policy in KeyNote

8.4.1 Identifying principals

For a detailed description refer to sections 4.2.1, 5.1.1 and 6.6. What we want to achieve is have a key signed by the key generation authority so that the signed key is a unique global identifier. If the principal requires an identity as is the case with human users and Kerberos principals then the certificate should indicate this identifier as well. Such a certificate has this form:

```
KeyNote-Version: 2
Comment: key generated by grid authentication service for
         user peter
Local-Constants:  AS_Key = "RSA:acdfa1df1011bbac"
                  Peter_Key = "DSA:deadbeefcafe001a"
Authorizer: AS_Key
Licensees: Peter_Key
Conditions: ((app_domain == "grid")  # valid for grid only
          && (identifier == "peter"));
Signature: "RSA-SHA1:f00f2244"
```

If the principals need no identifier, the identifier attribute is omitted.

8.4.2 Trusted Third Parties

For a detailed description refer to section 4.2.2.

```
KeyNote-Version: 2
Comment: Principal my_key is expressing trust in the third
         party ttp_key
Local-Constants:  my_key = "RSA:acdfa1df1011bbac"
                  ttp_Key = "DSA:deadbeefcafe001a"
Authorizer: my_Key
Licensees: ttp_Key
Conditions: ((app_domain == "grid")  # valid for grid only
          && (trust_level == "trusted"));
Signature: "RSA-SHA1:f00f2244"
```

8.4.3 Roles and Groups

For a detailed description refer to sections 4.2.3 and 4.2.4. Below are steps followed to create a group and define group members.

1. Create the key pair for the group and trust the public key. This is explained in section 8.4.2.
2. Create a membership certificate which essentially passes all future access rights to the members.

```
KeyNote-Version: "2"
Authorizer: "DSA:4401ff92"   # group key
Licensees: "DSA:abc991" ||   # group member 1 key
           "RSA:cde773" ||   # group member 2 key
           "BFIK:fd091a"     # group member 3 key
Conditions: (app_domain == "grid");
Signature: "DSA-SHA1:8912aa"
```

3. Assign access rights to the group and this invariably will give the rights to group members.

```
KeyNote-Version: "2"
Authorizer: "DSA:4401ff9252"   # group owner key
Licensees: "DSA:4401ff92"    # group key
Conditions: (app_domain == "grid" && write_access== "true" &&
             resource == "64764736");
Signature: "DSA-SHA1:8912aa"
```

8.4.4 Roles and Programs

For a detailed description of how roles can be utilized to securely load software refer to sections 4.2.3 and 4.2.5. To load a program named emacs securely, the node needs a certificate issued by node CA that binds a hash of the executable emacs.exe to the program name emacs. Let k-ca be the public key of the node CA and "emacs" be the program name. The CA can issue the following certificate.

```
KeyNote-Version: "2"
Authorizer: k-ca   # group owner key
Licensees: (object-hash(hash md5 |szKSlSK+SNzIsHH3wjAsTQ==|
                    emacs.exe)) # hash of executable program
Conditions: (app_domain == "grid" && program_name== "emacs");
Signature: "DSA-SHA1:8912aa"
```

Suppose emacs is a program written by a different user from CA and this user will continue to update emacs.exe. The proper thing would be for CA to delegate the right to certify emacs to this user so that the user can issue a certificate of the hash of new versions of emacs.exe. Let k-user be the public key for the user. The authorization certificate from CA to user would be:

```
KeyNote-Version: "2"
Authorizer: k-ca   # certifying authority key
Licensees: k-user  # program owner
Conditions: (app_domain == "grid" && programName== "emacs");
Signature: "DSA-SHA1:8912aa"
```

and the secure loading certificate from user for emacs.exe would be.

```
KeyNote-Version: "2"
Authorizer: k-user   # program owner key
Licensees: (object-hash(hash md5 |szKSlSK+SNzIsHH3wjAsTQ==|
                        emacs.exe)) # hash of executable program
Conditions: (app_domain == "grid" && program_name== "emacs");
Signature: "DSA-SHA1:8912aa"
```

8.4.5 Protecting Nodes

For a detailed description on how to determine that software is trusted or not in order to protect nodes executing it refer to section 4.2.12. The node CA creates a group called trustedSW as described in section 8.4.3. To make a program digest speak for the group (i.e trusted), the following certificate will suffice:

```
KeyNote-Version: "2"
Authorizer: k-trustedSW   # key for trustedSW group
Licensees: (object-hash(hash md5 |szKSlSK+SNzIsHH3wjAsTQ==|
                emacs.exe))  # hash of executable
Conditions: (app_domain == "grid" && trust_level == "trusted");
Signature: "DSA-SHA1:8912aa"
```

Alternatively, the CA can trust a user and allow user to issue trust certificates to programs. CA has to authorize the user to make these kind of certificates. CA creates a group called trustedSW-*owner* and makes the user a member of this group. Members of this group are trusted and so are any programs that they trust.

1.
```
KeyNote-Version: "2"
Authorizer: k-trustedSW-owner   # key for trustedSW-owner group
Licensees: k-user
Conditions: (app_domain == "grid" && trust_level == "trusted");
Signature: "DSA-SHA1:8912aa"
```

2.
```
KeyNote-Version: "2"
Authorizer: k-user   # key for program owner
Licensees: (object-hash(hash md5 |szKSlSK+SNzIsHH3wjAsTQ==|
                emacs.exe))  # hash of executable
Conditions: (app_domain == "grid" && trust_level == "trusted");
Signature: "DSA-SHA1:8912aa"
```

8.4.6 Delegation

For a detailed description refer to section 4.2.6.

8.4.6.1 Delegating Identity

A credential from RSA key abc1232323 unconditionally authorizing RSA key abc123 for all actions is essentially delegating identity to the holder of the secret key corresponding to abc123:

```
Authorizer: "RSA:abc1232323"
Licensees: "RSA:abc123"
```

8.4.6.2 Delegating rights

Conditions state limitations under which the authorization applies and are therefore used to limit access rights to specific rights as opposed to delegating identity described in previous section. Conditions are predicates that operate on the action attribute set to state delegated rights. This is a conclusion from the fact that the action attribute set state the security requirements of the application.

```
KeyNote-Version: 2
Comment: This credential gives access to a number of
principals. Write accesses have to be logged.
Authorizer: "RSA:dab212"      # Resource owner
Licensees: "DSA:feed1234" ||
          "RSA:abc123" ||
          "DSA:bcd987" ||
          "DSA:cde333" ||
          "DSA:def975" ||
          "DSA:978add"
Conditions: (app_domain="grid")  # nested clauses
              -> { (read_access == "true") -> _MAX_TRUST;
                 ( write_access == "true") -> "ApproveAndLog";};
Signature: "RSA-SHA1:186123"
```

8.4.6.3 Joint Delegation

User to node delegations (login, section 5.1.3) is a case of joint delegation in which the user (key k_u) is delegating to a short term login key (K_l) and to a long term key (k_w). To achieve this in KeyNote:

1. key k_u issues a 2-of-2 threshold certificate to K_l and k_w
   ```
   KeyNote-Version: 2
   Authorizer: "POLICY"
   Licensees: 2-of(k-l, k-w)
   Conditions: (app_domain=="grid" && resource == "64764737272");
   ```
2. Both must delegate to a common subject in order for the delegation to be valid. The node key k_w is the one that needs this delegation. Assuming that by default every key speaks for itself it is enough for k_l to delegate a short term key (refreshed every 30 minutes) to k_w. Without this assumption then k_w would need to delegate to itself. Delegation from k_l to k_w is as follows:
   ```
   KeyNote-Version: 2
   Authorizer: k-l
   Licensees: k-w
   Conditions: (app_domain=="grid" && start-date == "1999-03-01_12:00:00"
                && end-date == "1999-03-01_12:30:00");
   ```
   If k_l is forgotten (typically after user logout) and above delegation certificate expires the node can no longer speak for the user. The objective has been achieved that user-to-node delegation ends after user logs out.

8.4.7 Revoking Delegations

For detailed description refer to sections 4.2.8 and 5.10. KeyNote does not have any proposal for handling revocation of delegations. We have not yet formulated one.

8.4.8 Authorization

For a detailed description refer to sections 4.2.10, 4.2.9, 4.4.1 and 4.4.2.

8.4.8.1 Permissions Management

Below is a policy example that is unconditionally authorizing RSA key abc123 for all actions. It is equivalent to an ACL entry. This essentially defers the ability to specify policy to the holder of the secret key corresponding to abc123:

```
Authorizer: "POLICY"
Licensees: "RSA:abc123"
Conditions: (app_domain=="grid" && resource == "64764736");
```

If the ACL demands that a principal must speak for several principals in the ACL before granting access, such as have the right delegation and run from a trusted computer, then we use a threshold certificate in which the licensee is k-of-n. As an example according to the policy below, any two signers from the list of principals are sufficient to authorize the resource.

```
KeyNote-Version: 2
Authorizer: "POLICY"
Licensees: 2-of("DSA:feed1234", "DSA:978add", "DSA:978add")
Conditions: (app_domain=="grid" && resource == "64764736");
```

8.4.8.2 Authorization algorithm

KeyNote compliance checker finds and returns the Policy Compliance Value for queries as defined in [17].

8.5 Implementation Proposal

The implementation strategy and the objectives in this design are similar the to SPKI implementation proposal described in section 7.3. Figure 8.3 shows what the system components are in an implementation.

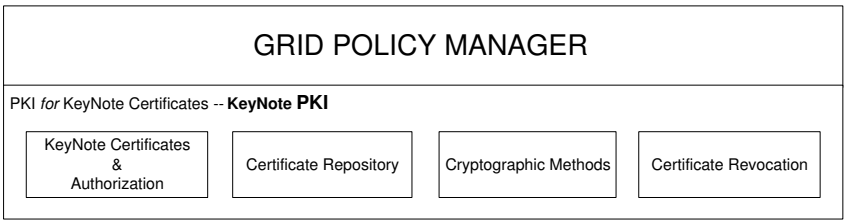

Figure 8.3: Components of a Grid Security System

8.5.1 Grid Policy Manager

The description and implementation proposal for the grid manager is the same one as in section 7.3.1.

8.5.2 KeyNote PKI

Figure 8.3 breaks down the KeyNote PKI into four sub-components. The KeyNote Certificate & Authorization component are implemented in the current implementation of KeyNote Version 2. The current release of this implementation is version 2.3. The source code for this release can be downloaded from [59].We recommend an implementation of KeyNote that runs as a trusted service that other applications communicate to using interprocess communications mechanisms.

The certificate repository and revocation components have neither an existing implementation nor a KeyNote proposal. Subsequently we have no implementation proposal for these two.

The description and implementation proposal for cryptographic methods is the same one as in section 7.3.4.

Chapter 9

GSI

9.1 What is GSI ?

Globus Security Infrastructure (GSI) is an implementation of a proposal for grid security architecture made by Foster et al. [49]. GSI was developed as part of the Globus project. GUSTO, a testbed that spans over 20 institutions and coupling over 2.5 teraflops of peak computing power, was built as part of Globus. GSI provides support for user proxies[1] (the Globus resource allocation manager (GRAM) [38]), resource proxies[2], certification authorities and implementation of the required protocols. All these are described in section 9.3.

9.2 Security approach

- The grid consists of multiple trust domains each with a local security solution that can neither be replaced nor be overridden. Grid solution must instead focus on controlling the interdomain interactions and mapping interdomain operations into local security policy.
- Local security can be implemented by a variety of methods including Kerberos, SSH, SSL e.t.c.
- Subjects have a local identity and a global identity. Local identities depend on the local security solution and global identities are based on the X.509 certificates. Each domain maintains a mapping of local-to-global identities for subjects using resources in that domain. This mapping is site specific. Some sites will map global names to predefined local names, or dynamic local names, or group names.
- The existence of global-subjects enable the policy to provide single sign-on.
- Operations between entities located in different trust domains require mutual authentication.
- An authenticated global subject mapped into a local subject is equivalent to locally authenticating the local subject in the local domain.
- All access control decisions are made locally on the basis of local subject.
- A program or process can act on behalf of a user and can be delegated a subset of the user's rights.
- Processes running on behalf of the same subject within the same trust domain may share a single set of credentials.

9.3 GSI architecture

The approach described in section 9.2 is the basis of the security policy in GSI. This policy provides the context to construct the architecture shown in figure 9.1. The architecture defines the set of subjects

[1] A user proxy is a session manager process given permission to act on behalf of a user for a limited period of time.

[2] A resource proxy is an agent used to translate between interdomain security operations and local intradomain mechanisms.

and objects under the jurisdiction of this policy and the protocols that govern interactions between these subjects and objects.

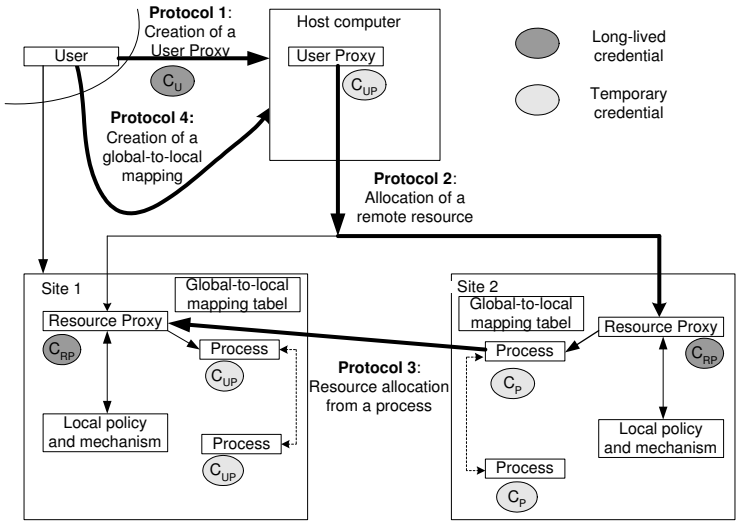

Figure 9.1: A computation grid security architecture

9.3.1 Entities

- The subjects are *users* and *processes* acting on behalf of users.
- The objects are the wide range of *resources* found in a grid environment: computers, data repositories, networks, display devices e.t.c. Grid computations grow and shrink dynamically, acquiring resources when need arises and releasing them when done with them. Processes obtain resources on behalf of users.
- Since it is impractical for a "user" to interact with each such resource for purpose of authentication, the user login starts a process, *user proxy*, that acts on behalf of the user. The user proxy acts on behalf of user for the duration of the computation or session.
- The architecture defines a *resource proxy*.

The architecture is determined by specifying the protocols that are used when subjects and objects interact. The interactions identified are:

- Creation of a process by a user.
- allocation of a resource by a process, and
- communication between processes located in different trust domains.

9.3.2 User Proxy Creation Protocol

This protocol generates a temporary credential C_{up} for the user proxy. The user gives permission to the proxy by signing this credential with a secret (e.g private key). C_{up} has a validity time interval as well as other restrictions imposed by the user such as hosts on which the proxy can operate and target sites where proxy is allowed to start processes and use resources. The user proxy can use this temporary credential to authenticate with resource proxies.

9.3.3 Resource Allocation Protocol

Resources are controlled by a resource proxy which is responsible for scheduling access to a resource and mapping a computation onto that resource. A user proxy that requires access to a resource will first

1. The user gains access to the computer from which the user proxy is to be created, using whatever form of local authentication required on that computer.
2. The user produces the user proxy credential, C_{up}, by using their credential, C_u, to sign a tuple containing the user's id, the name of the local host, the validity interval for C_{up}, and any other information that will be required by the authentication protocol used to implement the architecture (such as a public key if certificate-based authentication is used):
   C_{up} = Sig_u\{ user-id, host, start-time, auth-info, ... \}
3. A user proxy process is created and provided with C_{up}. It is up to the local security to protect the integrity of C_{up} on the computer on which the user proxy is located.

Table 9.1: Protocol 1 - User proxy creation

determine the identity of the resource proxy for that resource. It then issues a request to the resource proxy and if the request is successful, the resource is allocated and a process is created on that resource. Authorization is either performed at resource allocation time or process creation time depending on resource policy.

Protocol 2 defines the mechanism used to issue a request to a resource proxy from a user proxy. Step 3 in the protocol may require a global-to-local identity mapping if resource policy requires authorization at allocation time. This mapping is defined by protocol 4. Protocol 2 creates a temporary credential for the newly created process. This credential C_p gives the process a way to authenticate itself and gives it the identity of the user on whose behalf the process was created. All processes from a single resource allocation request possess the same credential.

1. The user proxy and resource proxy authenticate each other using C_{UP} and C_{RP}. As part of this process, the resource proxy checks to ensure that the user proxy's credentials have not expired.
2. The user proxy presents the resource proxy with a signed request in the form of $Sig_{UP}\{allocationspecification\}$.
3. The resource proxy checks to see whether the user who signed the proxy's credentials is authorized by local policy to make the allocation request.
4. If the request can be honored, the resource proxy creates a RESOURCE_CREDENTIALS tuple containing the name of the user for whom the resource is being allocated, the resource name, e.t.c.
5. The resource proxy securely passes the RESOURCE_CREDENTIALS to the user proxy. (This is possible from step 1.)
6. The user proxy examines the RESOURCE_CREDENTIALS request, and, if it wishes to approve, signs the tuple to produce C_P, a credential for the requesting resource.
7. The user proxy securely passes C_P to the resource proxy. (This is again possible due to step 1.)
8. The resource proxy allocates the resource and passes the new process(es) C_P. (This transfer relies on the fact that the resource proxy and process are in the same domain.)

Table 9.2: Protocol 2 - Resource allocation (and process creation)

9.3.4 Resource Allocation from a Process Protocol

Computations are started by a resource allocation from a user proxy. After this, successive resource allocations are initiated dynamically from a process created via a previous resource allocation request. Protocol 3 defines how this is accomplished. This technique used is not scalable [49] due to its reliance

1. The process and its user proxy authenticate each other using C_p and $C_u{-}p$.
2. The process issues a signed request to its user proxy, with the form:
   Sig_p { "allocate", allocation request parameters }
3. If the user process decides to honor the request, it initiates a resource allocation request to the specified resource proxy using protocol 2.
4. The resulting process handle is signed by the user proxy and returned to requesting process.

Table 9.3: Protocol 3 - Resource allocation from a user process

on a single user proxy but offers simplicity and fine-grained control.

9.3.5 Mapping Registration Protocol

The architecture relies on a mapping of global subjects to local subjects. Global name (e.g ticket or certificate) to local name (e.g login-name or user-id) pairs are stored in a *mapping table* maintained by the resource proxy. Mappings are added by users.

The basic idea behind this protocol is that a user has to prove that he holds credentials for both a local and a global subject. This is accomplished by authenticating both globally and directly to the resource using the local authentication method. The user then asserts a mapping between global and local credentials. The assertion is coordinated through the resource proxy, since it is in a position to accept both global and local credentials. Steps 1.a and 1.b are performed by user to authenticate globally. Steps 2.a and 2.b are performed by user to authenticate locally to the resource.

Matching MAP-SUBJECT-P and MAP-SUBJECT-UP requests must be issued from the user proxy and mapping process to ensure that the same user is in possession of both global and local credentials. If the result of the mapping is stored for continued access, the user will need to execute the mapping protocol once per resource. Duration of validity of the mapping depends on local policy. Part of the mapping protocol requires that the user logs into the resource for which the mapping is to be created which requires that user authenticate locally. Mapping is therefore as secure as the local authentication method.

1.a	User proxy authenticates with the resource proxy.
1.b	User proxy issues a signed MAP-SUBJECT-UP request to resource proxy, providing as arguments both global and local subject names.
2.a	User logs on to the resource using the resource's authentication method and starts a map registration process.
2.b	Map registration process issues MAP-SUBJECT-P request to resource proxy, providing as arguments both global and resource subject names.
1.	Resource proxy waits for MAP-SUBJECT-UP and MAP-SUBJECT-P requests with matching arguments.
2.	Resource proxy ensures that map registration process belongs to the resource subject specified in the map request.
3.	If a match is found, resource proxy sets up a mapping and sends acknowledgements to map registration process and user proxy.
4.	If a match is not found within MAP-TIMEOUT, resource proxy purges the outstanding request and sends an acknowledgement to the waiting entity.
5.	If acknowledgement is not received within MAP-TIMEOUT, request is considered to have failed.

Table 9.4: Protocol 4 - Mapping global to local identifier

9.4 GSI implementation

GSI provides support for user proxies, resource proxies (GRAM[3] [37]), certification authorities and implementation of specified protocols.

The architecture has specified protocols in terms of abstract operations rather than using any specific security technologies. This separation of protocol and mechanism is desirable in implementation too to support portability and flexibility of the resulting system. To achieve this separation, GSI is implemented on top of the Generic Security Services application programming interface (GSS-API) [65]. GSI-API provides services in a generic fashion, allowing implementation in a range of underlying mechanisms and technologies.

To construct GSI using GSS-API, the grid security protocols are simply transcribed into GSS calls. The relationship between Globus and GSS-API is shown in figure 9.2.

GSS-API is oriented to two-party security contexts. It provides functions for obtaining credentials, performing authentication, signing messages and encrypting messages. GSS-API is both *transport* and *mechanism* independent. This means that it does not depend on a specific communication method or library and it does not specify the use of specific security protocols such as Kerberos, SESAME e.t.c. GSI currently

[3] Globus Resource Allocation Manager

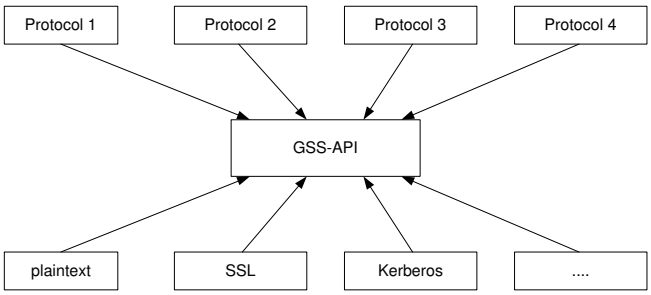

Figure 9.2: Use of GSS-API in Globus

uses raw TCP sockets and Nexus communication library [48] to move tokens between processors. GSS-API bindings have been defined for several mechanisms to date: a plaintext password-based mechanism and a X.509 certificate-based mechanism. GSS-API has been of significant benefit but it has the following limitations:

- *Lack of delegation:* Needed to allow temporary and limited transfer of user's rights to a process in the event that the user trusts the site (and resource) hosting this process enough to forgo an authentication/authorization handshake with the user proxy each time a new process needs to be created.
- *No support for group contexts:* Needed to support secure communication within a dynamic group of processes belonging to the same computation.

The GSI implementation currently uses the authentication protocols defined by the Secure Socket Library (SSL) protocol [50]. SSL defines two components: authentication and communication components. GSI uses GSS/SAP. GSS/SAP is an implementation of GSS that uses *SSL Authentication Protocol (SAP)*, a term used to refer to the authentication elements of SSL.

9.5 Comparison with proposed architecture

9.5.1 Requirements

GSI provides an analysis of the security problem in computation grids. This thesis has also undertaken a similar exercise and identified a set of requirements that are presented in chapter 3. The primary requirements identified are:Authentication, Authorization, Assurance, Accounting, Audit and Secure communication. GSI identifies the first two but not the rest.

The authentication and authorization requirements identified has also included additional requirements above those of GSI. These are:

- Data origin authentication.
- Authorization decisions have been expanded to consider not only the user requiring access but the node from which user is accessing the resource and the code that is to be executed in cases where users provide code.

We have not looked into export control regulation requirements because this is no longer relevant after recent policy changes in the US.

9.5.2 Policy

	Proposed Policy	**GSI Policy**
Identification and Authentication	•Principals are identified by key pairs. •Authentication is by proving knowledge of the private key.	•Principals have name identifiers •Authentication is by proving knowledge of a password.
Trusted Third Parties	•Trust relations are transitive. •Trust relations not based a CA. •Multiple levels of trust	•Trust relations not transitive. •Trust based on a global trusted CA •Binary trust decisions - Either trusted or not trusted.
Roles	•Principals can assert roles in order to reduce their authority.	•There is no support for roles at the inter-domain level.
Loading Programs	•There is support for secure loading of programs. This applies to both the OS as well as use programs.	•No secure loading of programs.
Delegation	•Processes can get user identity delegation as well as user rights delegation.	•Delegation of identity is only to the user proxy and not to processes. •There is no user rights delegation to processes. Every time that a process needs a resource an authentication/ authorization handshake with the user proxy has to take place before the user proxy can acquire resources for the process.
Administration Policy	•Allowed object accesses can be modified by object owner as well as those with delegations that permit them to do so.	•Allowed object access can only be modified by object owners.
Authorization Policy	Authorization is based on multiple parameters: •user making the request, •delegations to the user, •the node the user is logged into, •program and group roles asserted.	Authorization decision is based user identity only.

9.5.3 Architecture

	Proposed Architecture	**GSI Architecture**
Single-sign on	•Cascaded user-to-node, node-to-node, user-to-process and node-to-process delegations facilitate single sign on.	•Use of a user proxy addresses single sign-on requirement and avoids need to communicate user credentials. However, the use of a single user proxy for all resource allocations is a potential bottleneck.
Inter-operation with local security Solutions	•Use of TLS authentication protocol that supports a number of different credentials facilitate inter-operability with Kerberos domains. •Sharing keys between trusted third in different domains integrates the domains.	•Use of resource proxy enables inter-operability with local security solutions. Resource proxy translates between inter-domain and intra-domain security solutions.
Global-local Subject Mapping	•There is support for secure loading of programs. This applies to both the OS as well as use programs.	•No secure loading of programs.
Resource Allocation to Users	•User login delegates to the node which then makes resource requests on behalf of the user.	•User login delegates identity to the user proxy which makes request for resources on behalf of the user.
Resource Allocation to Processes	•User login delegates to the node. When a user makes a request to start a process the node starts the process and hands off to this process the ability to speak for the user The process can thereafter make requests for resources directly.	•A process that requires a resource makes a request for the resource to the user proxy. User proxy will then obtain the resource by making a request to the resource proxy. After receiving a resource handle, user proxy signs the handle and sends it to the process.

9.5.4 Implementation Mechanisms

	Proposed Implementation	**GSI Implementation**
Use of GSS-API	•We propose to use GSS-API once it has support for the required implementation mechanisms: KeyNote or SPKI certificates.	•Implementation of GSI utilizes the GSS-API. GSS-API is both transport and mechanism independent.
Transport Mechanisms	•Proposal is independent of transport mechanism.	•GSI currently uses TCP sockets and Nexus communication library.
Authentication Mechanisms	•Proposal is based on SPKI and Keynote certificates.	•GSI uses plaintext passwords and X.509 certificates.
Authentication Protocols	•Proposed architecture uses the authentication protocols defined by TLS.	•GSI uses the authentication protocols defined by SSL 3.0.

Chapter 10

Conclusion

10.1 Achievement

The primary achievement of this thesis has been an implementation proposal described in the following section. The byproducts of this proposal are an analysis of grid security requirements, a statement of a grid security policy using a formal notation, investigation of SPKI and KeyNote to determine the suitability of these mechanisms for enforcing security policy and their completeness in general.

10.2 Choice of Implementation Proposal

We recommend the use of SPKI over KeyNote. Primarily this is due to the weaknesses in the KeyNote proposal highlighted in section 8.3.4. The luck of proposals for certificate storage, certificate discovery and certificate revocation is critical since this functionality is necessary in an implementation that would enforce the proposed grid security policy. The SPKI implementation proposal is described in section 7.3.

10.3 Further Work

10.3.1 New Kerberos Authorization Mechanism

Look at how to incorporate Kerberos authorization proposal by Neumann [78]. This mechanism would solve some of the weaknesses of Kerberos authorization that are outlined in section6.7.

10.3.2 Cancelling Public Keys

There is occasionally a need to cancel a public key due to reasons that could include a compromise of the private key, withdrawal of service e.t.c. This would make any future use of the key invalid as well as cancel or transfer existing access rights. We have not outlined a policy for this in our grid. A number of issues exist here such as a parallel run of the two keys for sometime, transfer of access rights to the new key e.t.c. We have left this for as future work.

10.3.3 Asserting Roles

We have looked at how principals can make use of roles. What we have not looked at here is how principals assert these roles. In Linux, the *su* command is used to assert an administrator role. We need a somewhat similar mechanism for principals in the grid so that they can assert roles in a simple yet flexible manner.

10.3.4 Multiplexing Secure Channels

The need for multiplexing channels is described in section 5.2.4. The question here is how to multiplex channels into sub−channels. When two nodes first communicate they establish a node to node channel. We would like to multiplex this channel so that interprocess communication between the two nodes utilize this same channel.

10.3.5 Using Smart Cards

Smart cards have been proposed as a means of authenticating users to nodes. For a description of user to node authentication, refer to section 5.1.2.

The smart card stores the key pair and a user in possession of the smart card can therefore use it to authenticate to nodes. This is more secure than our current proposal of storing password-encrypted private keys on the Authentication Server. Abadi et al. [4] has such a proposal.

10.3.6 Online Certificate Tests

Online tests that return a revalidation object have been proposed as a way of implementing SPKI certificate revocation. In such a solution the authorization service makes a revalidation request to an online test service that is specified on the certificate. We have not specified the protocol for this interaction between the Authorization Service and the Online Test Service. Weise [99] suggests the use of Online Certificate Status Protocol (OCSP). We have not looked at this.

10.3.7 Authorizing Long Running Computations

The problem of long running computations is described in section 5.1.5 together with our proposed solution. There is however another solution to authorizing these computations suggested by Lampson et al. [64] that would be good to consider.

10.3.8 SPKI <tag> Object Structure

SPKI <tag> objects have been described in section 7.1.2. This field, when it exists in an authorization certificate, indicates the access rights that are delegated from the issuer to the subject. To define the structure of this field would require a detailed analysis of the rights being delegated in the grid.

10.3.9 Grid Operating Systems Software

Grid nodes have to run an operating systems. It will be ideal if one of existing operating systems can easily integrate our security system so that it is not necessary to develop a new OS. There are a number of operating systems that are available in the open source software domain and evaluating a number of these would determine which would be ideal for grid nodes.

10.3.10 Implementing a Prototype

A key extension to this thesis would be a prototype implementation of the SPKI implementation proposal.

10.4 Evaluation of Implementation Proposal

PKI has gained a great deal of interest going by the various number of commercials products available. A key challenge to adoption of PKI is storage and retrieval of the huge number of certificates that are generated in practice. With further improvements in this area, PKI should gain wider usage.

Appendix A

Glossary of TLS and Cryptography Terms

Below is a glossary of terms as defined in [40]

application protocol
An application protocol is a protocol that normally layers directly on top of the transport layer (e.g TCP/IP). Examples include HTTP, TELNET, FTP and SMTP

asymmetric cipher
see public key cryptography

block cipher
A block cipher is an algorithm that operates on plaintext in groups of bits, called blocks. 64 bits is a common block size.

bulk cipher
A symmetric encryption algorithm used to encrypt large quantities of data.

cipher block chaining (CBC)
CBC is a mode in which every plaintext block encrypted with a block cipher is first exclusive−ORed with the previous ciphertext block (or, in the case of the first block, with the initialization vector). For decryption, every block is first decrypted, then exclusive−ORed with the previous ciphertext block (or IV).

client
The application entity that initiates a TLS connection to a server. This may or may not imply that the client initiated the underlying transport connection. The primary operational difference between the server and client is that the server is generally authenticated, while the client is only optionally authenticated.

client write key
The key used to encrypt data written by the client.

client write MAC secret
The secret data used to authenticate data written by the client.

connection
A connection is a transport (in the OSI layering model definition)that provides a suitable type of

service. For TLS, such connections are peer to peer relationships. The connections are transient. Every connection is associated with one session.

Data Encryption Standard
DES is a very widely used symmetric key encryption algorithm. DES is a block cipher with a 56 bit key and an 8 byte block size. Note that in TLS, for key generation purposes, DES is treated as having an 8 byte key length (64 bits), but it still only provides 56 bit protection. (The low bit in each key bytes is presumed to be set to produce odd parity in that key byte.) DES can also be operated in a mode where three independent keys and three operations are used for each block of data; this uses 168 bits of key (24 bytes in the TLS key generation method)and provides the equivalent of 112 bits of security. [101], [94]

Digital Signature Standard
A standard for digital signing, including the Digital Signing Algorithm, approved by the National Institute of Standards and Technology, defined in NIST FIPS PUB 186, "Digital Signature Standard," published May, 1994 by the Us Dept. of Commerce [1].

handshake
An initial negotiation between client and server that establishes the parameters of their transactions.

Initialization Vector
When a block cipher is used in CBC mode, the initialization vector is exclusive−ORed with the first plaintext block prior to encryption.

IDEA
A 64−bit block cipher designed by Xuejia Lai and James Massey [63].

Message Authentication Code (MAC)
A Message Authentication Code is a one−way hash computed from a message and some secret data. It is difficult to forge without knowing the secret data. Its purpose is to detect if the message has been altered.

master secret
Secure secret data used for generating encryption encryption keys, MAC secrets, and IVs.

MD5
MD5 is a secure hashing function that converts an arbitrarily long data stream into a digest of fixed size (16 bytes). [85]

public key cryptography
A class of cryptographic techniques employing two−key ciphers. Messages encrypted with the public key can only be decrypted with the associated private key. Conversely, messages signed with the private key can be verified with the public key.

one−way hash function
A one−way transformation that converts an arbitrary amount of data into a fixed−length hash. It is computationally hard to reverse the transformation or to find collisions. MD5 and SHA are examples of one−way hash functions.

RC4
A stream cipher licensed by RSA Data Security. A compatible cipher is described in [86]

RSA
A widely used public−key algorithm that can be used for either encryption or digital signing. [26]

server
The server is the application entity that responds to requests for connections from clients. See also under client.

session
A TLS session is an association between a client and a server. Sessions are created by the handshake protocol. Sessions define a set of cryptographic security parameters, which can be shared among multiple connections. Sessions are used to avoid the expensive negotiation of new security parameters for each connection.

session identifier
A session identifier is a value generated by a server that identifies a particular session.

server write key
The key used to encrypt data written by the server.

server write MAC key
The secret data used to authenticate data written by the server.

SHA
The Secure Hash Algorithm is defined in FIPS PUB 180-1. It produces a 20−byte output. Note that all references to SHA actually use the modified SHA-1 algorithm. [82]

SSL
Netscape's Secure Socket Layer protocol [52]. TLS is based on SSL Version 3.0.

stream cipher
An encryption algorithm that converts a key into a cryptographically−strong keystream, which is then exclusive−ORed with the plaintext.

symmetric cipher
see bulk cipher.

Appendix B

Glossary of Kerberos Terms

Below is a glossary of Kerberos terms as defined in [95].

Authenticator
A record containing information that can be shown to have been recently generated using the session key known only by the client and server.

Capability
A token that grants the bearer permission to access an object or service. In Kerberos, this might be a ticket whose use is restricted by the contents of the authorization data field, but which lists no network addresses, together with the session key necessary to use the ticket.

Client
A process that makes use of a network service on behalf of a user. In some cases, a server may itself be a client of another server (e.g a print server may be a client of a file server).

Credentials
A ticket plus the secret session key necessary to successfully use that ticket in an authentication exchange.

KDC
Key Distribution Center, a network service that supplies tickets and temporary session keys; or an instance of that service or the host on which it runs. The KDC services both initial ticket and ticket−granting ticket requests. The initial ticket portion is sometimes referred to as the Authentication Server (or service). The ticket−granting ticket portion is sometimes referred to as the ticket−granting server (or service).

Secret key
An encryption key shared by the principal and the KDC, distributed outside the bounds of the system, with a long lifetime. In the case of a human user's principal, the secret key is derived from a password.

Server
A particular principal which provides a resource to network clients.

Service
A resource provided to network clients; often provided by more than one server (for example, remote file service).

Session key
A temporary encryption key used between two principals, with a lifetime limited to the duration of a single login "session".

Sub−session key
A temporary encryption key used between two principals, selected and exchanged by the principals using the session key, and with a lifetime limited to the duration of a single session.

Ticket
A record that helps a client authenticate itself to a server; it contains the client's identity, a session key, a timestamp, and other information, all sealed using the server's secret key. It only serves to authenticate a client when presented along with a fresh authenticator.

Appendix C

Glossary of SPKI Terms

These terms are as defined in [42]

1. ACL: an Access Control List: a list of entries that anchors a certificate chain. Sometimes called a "list of root keys", the ACL is the source of empowerment for certificates. That is, a certificate communicates power from its issuer to its subject, but the ACL is the source that power (since it theoretically has the owner of the resource being controlled as its implicit issuer). An ACL entry has potentially the same content as a certificate body, but has no Issuer (and is not signed). There is most likely one ACL for each resource owner, if not for each controlled resource.
2. CERTIFICATE: a signed instrument that empowers the Subject. It contains at least an Issuer and a Subject. It can contain validity conditions, authorization and delegation information. Certificates come in three categories: ID (mapping <name,key>), Attribute (mapping <authorization,name>), and Authorization (mapping <authorization,key>). An SPKI authorization or attribute certificate can pass along all the empowerment it has received from the Issuer or it can pass along only a portion of that empowerment.
3. CANONICAL S-EXPRESSION: an encoding of an S-expression that does not permit equivalent representations and is designed for easy parsing.
4. FULLY QUALIFIED NAME: a local name together with a global identifier defining the name space in which that local name is defined.
5. GLOBAL IDENTIFIER: a globally unique byte string, associated with the keyholder. In SPKI this is the public key itself, a collision-free hash of the public key or a Fully Qualified Name.
6. HASH: a cryptographically strong hash function, assumed to be collision resistant. In general, the hash of an object can be used wherever the object can appear. The hash serves as a name for the object from which it was computed.
7. ISSUER: the signer of a certificate and the source of empowerment that the certificate is communicating to the Subject.
8. KEYHOLDER: the person or other entity that owns and controls a given private key. This entity is said to be the keyholder of the keypair or just the public key, but control of the private key is assumed in all cases.
9. NAME: a SDSI name always relative to the definer of some name space. This is sometimes also referred to as a local name. A global (fully qualified) name includes the global identifier of the definer of the name space. For example, if
   (name jim)
   is a local name,
   (name (hash md5 |+gbUgUltGysNgewRwu/3hQ==|) jim)
   could be the corresponding fully qualified name.
10. ONLINE TEST: one of three forms of validity test: (1) CRL; (2) revalidation; or (3) one-time revalidation. Each refines the date range during which a given certificate or ACL entry is considered valid, although the last defines a validity interval of effectively zero length.

11. PRINCIPAL: a cryptographic key, capable of generating a digital signature. We deal with public-key signatures in this document but any digital signature method should apply.
12. PROVER: the entity that wishes access or that digitally signs a document. The Prover typically sends a message or opens a channel to the Verifier that then checks signatures and credentials sent by the Prover.
13. SPEAKING: A Principal is said to "speak" by means of a digital signature. The statement made is the signed object (often a certificate). The Principal is said to "speak for" the Keyholder.
14. SUBJECT: the thing empowered by a certificate or ACL entry. This can be in the form of a key, a name (with the understanding that the name is mapped by certificate to some key or other object), a hash of some object, or a set of keys arranged in a threshold function.
15. S-EXPRESSION: the data format chosen for SPKI/SDSI. This is a LISP- like parenthesized expression with the limitations that empty lists are not allowed and the first element in any S-expression must be a string, called the "type" of the expression.
16. THRESHOLD SUBJECT: a Subject for an ACL entry or certificate that specifies K of N other Subjects. Conceptually, the power being transmitted to the Subject by the ACL entry or certificate is transmitted in (1/K) amount to each listed subordinate Subject. K of those subordinate Subjects must agree (by delegating their shares along to the same object or key) for that power to be passed along. This mechanism introduces fault tolerance and is especially useful in an ACL entry, providing fault tolerance for "root keys".
17. TUPLE: The security-relevant fields from a certificate or ACL entry. We speak of 4-tuples for name certificates and 5-tuples for authorizations. The 4-tuple has fields:
    <Issuer, Name, Subject, Validity>
    while the 5-tuple has fields:
    <Issuer, Subject, Delegation, Authorization, Validity>.
18. VALIDITY CONDITIONS: a date range that must include the current date and time and/or a set of online tests that must succeed before a certificate is to be considered valid.
19. VERIFIER: the entity that processes requests from a prover, including certificates. The verifier uses its own ACL entries plus certificates provided by the prover to perform "5-tuple reduction", to arrive at a 5-tuple it believes about the prover: <self,prover,D,A,V>.

Appendix D

BNF definition of SPKI objects

The following is the BNF of canonical forms and includes lengths for each explicit byte string. So, for example, "cert" is expressed as "4:cert". These terms are as defined in [42]

D.1 Top Level Objects

The list of BNF rules that follows is sorted alphabetically, not grouped by kind of definition. The top level objects defined are:

<5-tuple>: an object defined for documentation purposes only. The actual contents of a 5-tuple are implementation dependent.
<acl>: an object for local use which might be implementation dependent. An ACL is not expected to be communicated from machine to machine.
<crl>, <delta-crl> and <reval>: objects returned from online tests.
<sequence>: the object carrying keys, certificates and online test results from prover to verifier.

D.2 Alphabetical List of BNF Rules

```
<5-tuple>:: <issuer5> <subject5> <deleg5> <tag-body5> <valid5> ;
<acl-entry>:: "(" "entry" <subj-obj> <deleg>? <tag> <valid>?
<comment>? ")" ;
<acl>:: "(" "acl" <version>? <acl-entry>* ")" ;
<byte-string>:: <bytes> | <display-type> <bytes> ;
<bytes>:: <decimal> ":" {binary byte string of that length} ;
<cert-display>:: "(" "display" <byte-string> ")" ;
<cert>:: "(" "cert" <version>? <cert-display>? <issuer> <issuer-loc>?
<subject> <subject-loc>? <deleg>? <tag> <valid>? <comment>? ")" ;
<comment>::  "(" "comment" <byte-string> ")" ;
<crl>:: "(" "crl" <version>? <crl-hash-list> <valid-basic> ")" ;
<crl-hash-list>:: "(" "canceled" <hash>* ")" ;
<date>:: <byte-string> ;
<ddigit>:: "0" | <nzddigit> ;
<decimal>:: <nzddigit> <ddigit>* | "0" ;
<deleg5>:: "t" | "f" ;
<deleg>:: "(" "propagate" ")" ;
<delta-crl>:: "(" "delta-crl" <version>? <hash-of-crl> <crl-hash-
```

```
list> <valid-basic> ")" ;
<display-type>:: "[" <bytes> "]" ;
<dsa-sig-val>:: "(" "dsa-sha1-sig" "(" "r" <byte-string> ")" "(" "s"
<byte-string> ")" ")" ;
<fq-name>:: "(" "name" <principal> <names> ")" ;
<general-op>:: "(" "do" <byte-string> <s-part>* ")" ;
<gte>:: "g" | "ge" ;
<hash-alg-name>:: "md5" | "sha1" | <uri> ;
<hash-list>:: "(" "canceled" <hash>* ")" ;
<hash-of-crl>:: <hash> ;
<hash-of-key>:: <hash> ;
<hash-op>:: "(" "do" "hash" <hash-alg-name> ")" ;
<hash-value>:: <byte-string> ;
<hash>:: "(" "hash" <hash-alg-name> <hash-value> <uris>? ")" ;
<integer>:: <byte-string> ;
<issuer-loc>:: "(" "issuer-info" <uris> ")" ;
<issuer-name>:: "(" "issuer" "(" "name" <principal> <byte-string> ")"
")" ;
<issuer5>:: <key5> | "self" ;
<issuer>:: "(" "issuer" <principal> ")" ;
<k-val>:: <integer> ;
<key5>:: <pub-key> ;
<keyholder-obj>:: <principal> | <name> ;
<keyholder>:: "(" "keyholder" <keyholder-obj> ")" ;
<low-lim>:: <gte> <byte-string> ;
<lte>:: "l" | "le" ;
<n-val>:: <integer> ;
<name-cert>:: "(" "cert" <version>? <cert-display>? <issuer-name>
<subject> <valid>? <comment>? ")" ;
<name>:: <relative-name> | <fq-name> ;
<names>:: <byte-string>+ ;
<not-after>:: "(" "not-after" <date> ")" ;
<not-before>:: "(" "not-before" <date> ")" ;
<nzddigit>:: "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9" ;
<obj-hash>:: "(" "object-hash" <hash> ")" ;
<one-valid>:: "(" "one-time" <byte-string> ")" ;
<online-test>:: "(" "online" <online-type> <uris> <principal> <s-
part>* ")" ;
<online-type>:: "crl" | "reval" | "one-time" ;
<op>:: <hash-op> | <general-op> ;
<principal>:: <pub-key> | <hash-of-key> ;
<pub-key>:: "(" "public-key" "(" <pub-sig-alg-id> <s-expr>* ")"
<uris>? ")" ;
<pub-sig-alg-id>:: "rsa-pkcs1-md5" | "rsa-pkcs1-sha1" | "rsa-pkcs1" |
"dsa-sha1" | <uri> ;
<range-ordering>:: "alpha" | "numeric" | "time" | "binary" | "date" ;
<relative-name>:: "(" "name" <names> ")" ;
<reval-body>:: <one-valid> | <valid-basic> ;
<reval-hash-list>:: "(" "valid" <hash>+ ")" ;
<reval>:: "(" "reval" <version>? <reval-hash-list> <reval-body> ")" ;
<s-expr>:: "(" <byte-string> <s-part>* ")" ;
<s-part>:: <byte-string> | <s-expr> ;
<seq-ent>:: <cert> | <name-cert> | <pub-key> | <signature> | <op> |
<reval> | <crl> | <delta-crl> ;
```

```
<sequence>:: "(" "sequence" <seq-ent>* ")" ;
<sig-params>:: <byte-string> | <s-expr>+ ;
<sig-val>:: "(" <pub-sig-alg-id> <sig-params> ")" ;
<signature>:: "(" "signature" <hash> <principal> <sig-val> ")" ;
<simple-tag>:: "(" <byte-string> <tag-expr>* ")" ;
<subj-hash>:: "(" "cert" <hash> ")" ;
<subj-obj>:: <principal> | <name> | <obj-hash> | <keyholder> | <subj-
thresh> ;
<subj-thresh>:: "(" "k-of-n" <k-val> <n-val> <subj-obj>* ")" ;
<subject-loc>:: "(" "subject-info" <uris> ")" ;
<subject5>:: <key5> | <fq-name5> | <obj-hash> | <keyholder> | <subj-
thresh> ;
<subject>:: "(" "subject" <subj-obj> ")" ;
<tag-body5>:: <tag-expr> | "null" ;
<tag-expr>:: <simple-tag> | <tag-set> | <tag-string> ;
<tag-prefix>:: "(" "*" "prefix" <byte-string> ")" ;
<tag-range>:: "(" "*" "range" <range-ordering> <low-lim>? <up-lim>?
")" ;
<tag-set>:: "(" "*" "set" <tag-expr>* ")" ;
<tag-star>:: "(" "tag" "(*)" ")" ;
<tag-string>:: <byte-string> | <tag-range> | <tag-prefix> ;
<tag>:: <tag-star> | "(" "tag" <tag-expr>  ")" ;
<up-lim>:: <lte> <byte-string> ;
<uri>:: <byte-string> ;
<uris>:: "(" "uri" <uri>+ ")" ;
<valid-basic>:: <not-before>? <not-after>? ;
<valid5>:: <valid-basic> | "null" | "now" ;
<valid>:: "(" "valid" <valid-basic> <online-test>* ")" ;
<version>:: "(" "version" <integer> ")" ;
```

Appendix E

Sample KeyNote Assertions

E.1 Traditional CA/Email

A. A policy unconditionally authorizing RSA key abc123 for all actions. This essentially defers the ability to specify policy to the holder of the secret key corresponding to abc123:

```
Authorizer: "POLICY"
Licensees: "RSA:abc123"
```

B. A credential assertion in which RSA Key abc123 trusts either RSA key 4401ff92 (called 'Alice') or DSA key d1234f (called 'Bob') to perform actions in which the "app_domain" is "RFC822-EMAIL", where the "address" matches the regular expression "^.*@keynote\.research\.att\.com$". In other words, abc123 trusts Alice and Bob as certification authorities for the keynote.research.att.com domain.

```
KeyNote-Version: 2
Local-Constants: Alice="DSA:4401ff92"  # Alice's key
                 Bob="RSA:d1234f"      # Bob's key
Authorizer: "RSA:abc123"
Licensees: Alice || Bob
Conditions: (app_domain == "RFC822-EMAIL") &&
            (address ~=   # only applies to one domain
              "^.*@keynote\\.research\\.att\\.com$");
Signature: "RSA-SHA1:213354f9"
```

C. A certificate credential for a specific user whose email address is mab@keynote.research.att.com and whose name, if present, must be "M. Blaze". The credential was issued by the 'Alice' authority (whose key is certified in Example B above):

```
KeyNote-Version: 2
Authorizer: "DSA:4401ff92"  # the Alice CA
Licensees: "DSA:12340987"   # mab's key
Conditions: ((app_domain == "RFC822-EMAIL") &&
             (name == "M. Blaze" || name == "") &&
             (address == "mab@keynote.research.att.com"));
Signature: "DSA-SHA1:ab23487"
```

D. Another certificate credential for a specific user, also

```
issued by the 'Alice' authority.  This example allows three
different keys to sign as jf@keynote.research.att.com (each
for a different cryptographic algorithm).  This is, in
effect, three credentials in one:
  KeyNote-Version: "2"
  Authorizer: "DSA:4401ff92"   # the Alice CA
  Licensees: "DSA:abc991" ||   # jf's DSA key
             "RSA:cde773" ||   # jf's RSA key
             "BFIK:fd091a"     # jf's BFIK key
  Conditions: ((app_domain == "RFC822-EMAIL") &&
               (name == "J. Feigenbaum" || name == "") &&
               (address == "jf@keynote.research.att.com"));
  Signature: "DSA-SHA1:8912aa"
Observe that under policy A and credentials B, C and D, the
following action attribute sets are accepted (they return
_MAX_TRUST):
    _ACTION_AUTHORIZERS = "dsa:12340987"
    app_domain = "RFC822-EMAIL"
    address = "mab@keynote.research.att.com"
 and
    _ACTION_AUTHORIZERS = "dsa:12340987"
    app_domain = "RFC822-EMAIL"
    address = "mab@keynote.research.att.com"
    name = "M. Blaze"
while the following are not accepted (they return
_MIN_TRUST):
    _ACTION_AUTHORIZERS = "dsa:12340987"
    app_domain = "RFC822-EMAIL"
    address = "angelos@dsl.cis.upenn.edu"
 and
    _ACTION_AUTHORIZERS = "dsa:abc991"
    app_domain = "RFC822-EMAIL"
    address = "mab@keynote.research.att.com"
    name = "M. Blaze"
 and
    _ACTION_AUTHORIZERS = "dsa:12340987"
    app_domain = "RFC822-EMAIL"
    address = "mab@keynote.research.att.com"
    name = "J. Feigenbaum"
```

E.2 Work Flow/Electronic Commerce

```
E. A policy that delegates authority for the "SPEND" application
   domain to RSA key dab212 when the amount given in the
   "dollars" attribute is less than 10000.
     Authorizer: "POLICY"
     Licensees: "RSA:dab212"  # the CFO's key
     Conditions: (app_domain=="SPEND") && (@dollars < 10000);
F. RSA key dab212 delegates authorization to any two signers,
   from a list, one of which must be DSA key feed1234 in the
   "SPEND" application when @dollars < 7500.  If the amount in
   @dollars is 2500 or greater, the request is approved but
```

```
   logged.

  KeyNote-Version: 2 Comment: This credential specifies a spending
  policy Authorizer: "RSA:dab212" # the CFO
  Licensees: "DSA:feed1234" && # The vice president
              "RSA:abc123" || # middle manager #1
              "DSA:bcd987" || # middle manager #2
              "DSA:cde333" || # middle manager #3
              "DSA:def975" || # middle manager #4
              "DSA:978add")   # middle manager #5
  Conditions: (app_domain=="SPEND") # note nested clauses
                  -> {(@(dollars) < 2500) -> _MAX_TRUST;
                      (@(dollars) < 7500) -> "ApproveAndLog"; };
  Signature: "RSA-SHA1:9867a1"

G. According to this policy, any two signers from the list of
   managers will do if @(dollars) < 1000:
     KeyNote-Version: 2
     Authorizer: "POLICY"
     Licensees: 2-of("DSA:feed1234", # The VP
                     "RSA:abc123",   # Middle management clones
                     "DSA:bcd987",
                     "DSA:cde333",
                     "DSA:def975",
                     "DSA:978add")
     Conditions: (app_domain=="SPEND") &&
                 (@(dollars) < 1000);
H. A credential from dab212 with a similar policy, but only one
   signer is required if @(dollars) < 500.  A log entry is made if
   the amount is at least 100.
     KeyNote-Version: 2
     Comment: This one credential is equivalent to six separate
              credentials, one for each VP and middle manager.
              Individually, they can spend up to $500, but if
              it's $100 or more, we log it.
     Authorizer: "RSA:dab212"      # From the CFO
     Licensees: "DSA:feed1234" ||  # The VP
                "RSA:abc123" ||    # The middle management clones
                "DSA:bcd987" ||
                "DSA:cde333" ||
                "DSA:def975" ||
                "DSA:978add"
     Conditions: (app_domain="SPEND")  # nested clauses
                   -> { (@(dollars) < 100) -> _MAX_TRUST;
                        (@(dollars) < 500) -> "ApproveAndLog";
                      };
     Signature: "RSA-SHA1:186123"
Assume a query in which the ordered set of Compliance Values is
{"Reject", "ApproveAndLog", "Approve"}.  Under policies E and G,
and credentials F and H, the Policy Compliance Value is
"Approve" (_MAX_TRUST) when:
     _ACTION_AUTHORIZERS = "DSA:978add"
     app_domain = "SPEND"
     dollars = "45"
```

```
     unmentioned_attribute = "whatever"
 and
     _ACTION_AUTHORIZERS = "RSA:abc123,DSA:cde333"
     app_domain = "SPEND"
     dollars = "550"
The following return "ApproveAndLog":
     _ACTION_AUTHORIZERS = "DSA:feed1234,DSA:cde333"
     app_domain = "SPEND"
     dollars = "5500"
 and
     _ACTION_AUTHORIZERS = "DSA:cde333"
     app_domain = "SPEND"
     dollars = "150"
However, the following return "Reject" (_MIN_TRUST):
     _ACTION_AUTHORIZERS = "DSA:def975"
     app_domain = "SPEND"
     dollars = "550"
 and
     _ACTION_AUTHORIZERS = "DSA:cde333,DSA:978add"
     app_domain = "SPEND"
     dollars = "5500"
```

Bibliography

[1] NIST FIPS PUB 166, *Digital signature standard*, Tech. report, National Institute of Standards and Technology, U.S Dept. of Commerce, May 1994.

[2] Eastlake 3rd and D. Gudmundsson, *Storing certificates in the domain name system*, Tech. Report RFC 2538, The Internet Society, November 1997.

[3] Eastlake 3rd and D. Kaufman, *Domain name system security extensions*, Tech. report, IETF Working Group, 1997.

[4] Abadi, M. Burrows, M. Kaufman, and B. Lampson, *Authentication and delegation with smart-cards*, Tech. report, DEC Systems Research Center, October 1990.

[5] M. Abadi, *On SDSI's linked local name spaces*, PCSFW: Proceedings of The 10th Computer Security Foundations Workshop, IEEE Computer Society Press, December 1997.

[6] Martín Abadi, Michael Burrows, Butler Lampson, and Gordon Plotkin, *A calculus for access control in distributed systems*, ACM Transactions on Programming Languages and Systems **15** (1993), no. 4, 706–734.

[7] S. Ames, M. Gasser, and R. Schell, *Security kernel design and implementation: An introduction*, IEEE Computer **16** (1983), 14–22.

[8] Ronny Havard Arild, *Delegation of authority in a multi-domain distributed file repository*, Master's thesis, University of Tromsø, November 1998.

[9] Tuomas Aura, *On the structure of delegation networks*, In proc of 11th IEEE Computer Security Foundations Workshop (Rockport, Massachusetts), June 1998, pp. 14–26.

[10] ———, *Distributed access-rights managements with delegations certificates*, Secure Internet Programming, 1999, pp. 211–235.

[11] Tuomas Aura, Petteri Koponen, and Juhana Rasanen, *Delegation-based access control for intelligent network services*, In proc of ECOOP Workshop on Distributed Object Security (Brussels, Belgium), July 1998.

[12] F. Ballesteros and L. Fernandez, *An adaptable and extensible framework for distributed object management*, In Special Issues in Object Oriented Programming. Workshop reader of the 10th European Conference on Object-Oriented Programming, ECOOP'96. (Linz(Au)), July 1996.

[13] S M Bellovin and M Merritt, *Limitations of the kerberos authentication system*, Tech. Report 5, AT&T Bell Laboratories, 1990.

[14] Matt Blaz, *Using the keynote trust management system*, http://www.crypto.com/trustmgt/.

[15] M. Blaze, J. Ioannidis, and A. Keromytis, *Delegation-based access control for intelligent network services*, In Proceedings of the Network and Distributed System Security Symposium (NDSS 2001) (San Diego, California), February 2001.

[16] Matt Blaze, Joan Fegenbaum, John Ioannidis, Angelo Keromytis, Jack Lacy, and Martin Strauss, *Trust management, compliance checking & security policy*, Policy 2001 Conference, Bristol UK, http://www.crypto.com/talks.

[17] Matt Blaze, Joan Feigenbaum, John Ioannidis, and Angelos D. Keromytis, *The keynote trust-management system version 2*, Tech. Report RFC 2704, IETF, September 1999.

[18] Matt Blaze, Joan Feigenbaum, and Angelos D. Keromytis, *Keynote: Trust management for public-key infrastructures (position paper)*, Security Protocols Workshop, 1998, pp. 59–63.

[19] ———, *The role of trust management in distributed systems security*, Secure Internet Programming, 1999, pp. 185–210.

[20] Matt Blaze, Joan Feigenbaum, and Jack Lacy, *Decentralized trust management*, Tech. Report 96-17, AT&T Labs and Distributed Systems Lab-University of Pennsylvania, 28, 1996.

[21] Matt Blaze, Joan Feigenbaum, and Martin Strauss, *Compliance checking in the policymaker trust management system*, Financial Cryptography, 1998, pp. 254–274.

[22] Matt Blaze, John Ioannidis, and Angelos D. Keromytis, *Keynote:trust management for public-key infrastructures*, Tech. report, AT & T Labs - Research and University of Pennsylvania.

[23] ———, *Offline micropayments without trusted hardware*, Financials Cryptography 2001, January 2001.

[24] ———, *Trust management for ipsec*, Proceedings of the Network and Distributed System Security Symposium (NDSS 2001) (San Diego, California), February 2001.

[25] Martin Abadi Michael Burrows and Roger Needham, *A logic of authentication*, Proceedings of the Royal Society of London (Litchfield Park, Arizona), 1990, pp. 18–36.

[26] R. Butler, D. Engert, I. Foster, C. Kesselman, S. Tuecke, J. Volmer, and V. Welch, *A method for obtaining digital signatures and public-key cryptosystems*, Communications of the ACM **21** (1978), 120–126.

[27] ———, *Design and deployment of a national-scale authentication infrastructure*, IEEE Computer **33** (1999), no. 12, 60–66.

[28] Ellison C., *Generalised certificates*, http://stud4.tuwien.ac.at/e9001175/, September 1999.

[29] Laferrier C. and Charland, *Authentication and authorization techniques in distributed systems*, Proceedings of IEEE International Carnahan Conference on Security Technology, 1993, pp. 164–170.

[30] Thaker C., Stewart L., and Satterthwaite E., *Firefly: A multiprocessor workstation.*, IEEE Trans. Computers **37** (1988), no. 8, 909–920.

[31] J. Casazza, *The development of electric power transmission: The role played by technology, institutions and people*, IEEE Computer Society Press, 1993.

[32] National Computer Security Center, *A guide to understanding discretionary access control in trusted systems*, Tech. Report Technical Report NCSC-TG-003 version-1, National Computer Security Center, Fort George G. meade, Maryland, September 1987.

[33] Certificate Discovery Using SPKI/SDSI 2.0 Certificates, *J. elien*, Master's thesis, Massachusetts Institute of Technology, May 1998.

[34] D. Chadwick, A. Young, and N. Cicovic, *Merging and extending the pgp and pem trust models – the ice-tel trust model*, IEEE Network **11** (1997), no. 3, 16–24.

[35] Santosh Chokhani, *Toward a national public key infrastructure*, IEEE Communications MAGAZINE **32** (1994), no. 9, 70–74.

[36] National Computer Security Control, *A guide to understanding audit in trusted systems*, Tech. Report NCSC-TG-001, Version 2, National Computer Security Control, June 1998.

[37] K. Czajkowski, I. Foster, N. Karonis, C. Kesselman, S. Martin, W. Smith, and S. Tuecke, *A resource management architecture for metacomputing systems*, IPPS/SPDP '98 Workshop on Job Scheduling Strategies for Parallel Processing, 1998.

[38] Karl Czajkowski, Ian T. Foster, Nicholas T. Karonis, Carl Kesselman, Stuart Martin, Warren Smith, and Steven Tuecke, *A resource management architecture for metacomputing systems*, JSSPP, 1998, pp. 62–82.

[39] Redell D.D, *Naming and protection in exetensible operating systems*, 9 ed., M.I.T Press, Cambridge MA, November 1974.

[40] T. Dierks, Certicom, and C. Allen, *The tls protocol version 1.0*, Tech. Report RFC 2246, Network Working Group, January 1999.

[41] Carl Ellison, Bill Frantz, Butler Lampson, Ron Rivest, Brian Thomas, and Tatu Ylonen, *Spki examples*, Internet Draft (Work in Progress), March 1998, draft-ietf-spki-cert-examples-01.txt.

[42] ———, *Simple public key certificate*, Internet Draft (Work in Progress), July 1999, http://world.std.com/cme/spki.txt.

[43] Carl M. Ellison, Bill Frantz, Butler Lampson, Ron Rivest, Brian M. Thomas, and Tatu Ylonen, *SPKI certificate theory*, Tech. Report RFC 2693, The Internet Society, September 1999.

[44] Internet Engineering Task Force, *Grid security infrastructure (gsi) roadmap*, Internet Draft, July 2001, Expires July 2001.

[45] I. Foster and C. Kesselman, *Computational Grids: The Future of High Performance Distributed*

Computing, Morgan Kaufmann, New York, 1998.

[46] ———, *The globus project: A status report*, 7th IEEE Heterogeneous Computing Workshop, March 1998, pp. 4–18.

[47] ———, *The Grid: Blueprint for a New Computing Infrastructure*, Morgan Kaufmann Inc., California, 1998.

[48] I. Foster, C. Kesselman, and S. Tuecke, *The nexus approach to integrating multithreading and communication*, Journal of Parallel and Distributed Computing **37** (1996), no. 1, 70–82.

[49] Ian Foster, Carl Kesselman, Gene Tsudik, and Steven Tuecke, *A security architecture for computational grids*, ACM Conference on Computer and Communications Security, 1998, pp. 83–92.

[50] Alan O. Freier, Philip Karlton, and Paul C. Kocher, *The ssl protocol version 3.0*, Internet draft (draft-freier-sslversion3-02.txt), November 1996, Work in Progress.

[51] A. Frier, P. Karlton, and P. Kocher, 1996.

[52] ———, *The ssl 3.0 protocol*, Tech. report, Netscape Communications Corp., November 1996.

[53] R. Ganesan, *Yaksha: Augmenting kerberos with public-key cryptography*, In Proceedings of the Internet Society Symposium on Network and Distributed System Security, February 1995, pp. 132–143.

[54] M. Gasser, A. Goldstein, C. Kaufman, and B. Lampson, *The digital distributed system security architecture*, In proc of 11th IEEE Computer Security Foundations Workshop (Baltimore, MD USA), National Institute of Standards and Technology (NIST), National Computer Security Center (NCSC)., October 1989, pp. 305–319.

[55] Morrie Gasser and E. McDermott, *An architecture for practical delegation in a distributed system*, IEEE Symposium on Security and Privacy, 1990, pp. 20–30.

[56] Mathew Hur, Joseph Salowey, and Ari Medvinsky, *Kerberos cipher suites in transport layer security*, Tech. Report Internet Draft, Network Working Group, November 2001.

[57] K. Jackson, LBNL, S. Tuecke, D. Engert, and ANL, *Tls delegation protocol*, Internet Draft, February 2001, Expires February 2002.

[58] T. Jaeger, *Access control in configurable systems*, Lecture Notes in Computer Science **1603** (1999), 289–316.

[59] Angelos Keromytis, *The keynote trust-management system*, http://www.cis.upenn.edu/ keynote/index.html.

[60] J. Kohl and B. Neuman, *The kerberos network authentication service v5*, RFC 1510, September 1993, p. 112.

[61] J. Kohl and C. Neuman, *The kerberos network authentication service v5*, Tech. Report RFC 1510, Network Working Group, September 1993.

[62] John T. Kohl, B. Clifford Neuman, and Theodore Y.Ts'o, *Distributed open systems*, ch. The Evolution of the Kerberos Authentication Service, pp. 78–94, IEEE Computer Society Press, 1994.

[63] X. Lai, *On the design and security of block ciphers*, ETH Series in Information Processing (1992).

[64] B. Lampson, M. Abadi, M. Burrows, and E. Wobber, *Authentication in distributed systems: Theory and practice.*, ACM Transactions on Computer Systems **10** (1992), no. 4, 265–310.

[65] J. Linn, *Generic security service application program interface*, Tech. Report RFC 1508 DRAFT, The Internet Society, 1990.

[66] ———, *Generic security service application program interface, version 2*, Tech. Report RFC 2078, Network Working Group, January 1997, Obsoletes: 1508.

[67] E. C. Lupu, D. A. Marriott, M. S. Sloman, and N. Yialelis, *A policy based role framework for access control*, In First ACM/NIST Role Based Access Control Workshop, December 1995.

[68] Sirbu M.A. and J.C.-I. Chuang, *Distributed authentication in kerberos using public key cryptography*, Symposium on Network and Distributed System Security (San Diego, California:), IEEE Computer Society Press., 1997.

[69] P. McMahon, *Sesame v2 public key and authorization extensions to kerberos*, In Proceedings of the 1995 Symposium on Netwrok and Distributed System Security, February 1995, pp. 114–131.

[70] A. Medvinsky and M. Hur, *Addition of kerberos cipher suites to transport layer security (tls)*, Tech. Report RFC 2712, Network Working Group, October 1999.

[71] Sun Microsystems, *Javatm cryptography extension (jce) reference guide for the javatm 2 sdk, standard edition, v 1.4*, Tech. report, Sun Microsystems.

[72] P. Mockapetris, *Domain names - concepts and facilities*, Tech. Report RFC-1034, Internet Network

Information., November, 1987.

[73] Sape J. Mullender, Andrew S. Tanenbaum, and Robbert van Renesse, *Using sparse capabilities in a distributed operating system*, pp. 103–114, Centrum voor Wiskunde en Informatica, Amsterdam, 1986.

[74] "Per Harald Myrvang", *"an infrastructure for authentication, authorization and delegation"*, Master's thesis, "University of Tromsø", "May" "2000".

[75] Needham and M. Schroeder, *Using encryption for authentication in large networks of computers*, Communications of the ACM **21** (1978), no. 12, 003–999.

[76] R. Needham, *Distributed systems*, 2nd edition ed., ch. Cryptography and Secure Channels, pp. 531–541, ACM Press, 1993.

[77] B. Neumanand and T. Ts'o, *Kerberos: An authentication service for computer networks*, IEEE Communications **32** (1994), no. 9, 33–38.

[78] B. Neumann, *Proxy-based authorization and accounting for distributed systems*, in Proceedings of the 13th International Conference on Distributed Computing Systems (Pittsburgh, PA), May 1993.

[79] P. Nikander, *An architecture for authorization and delegation in distributed object-oriented agent systems*, Ph.D. thesis, Helsinki University of Technology, March 1999.

[80] P. Nikander and L. Viljanen, *Storing and retrieving internet certificates.*, Proceedings of the 3rd Nordic Workshop on Secure IT Systems (NordSec '98) (Trondheim, Norway), November 1998.

[81] National Bureau of Standards, *Data encryption standard*, Federal Information Processing Standards Pub. **46** (1977).

[82] National Institute of Standards and U.S Dept. of Commerce Technology, *Secure hash standard*, Work in Progress, May 1994.

[83] "Jonna Partanen", *"using spki certificates for access control in java 1.2"*, Master's thesis, "Helsinki University of Technology", "August" "1998".

[84] Needham R., *Distributed systems*, 2nd ed., ch. Cryptography and Secure channels, pp. 231–241, ACM Press, 1993.

[85] Rivest R., *The md5 message digest algorithm*, RFC 1321, April 1992.

[86] Thayer R. and Kaukonen K., *A stream cipher encryption algorithm*, Internet-Draft draft-kaukonen-cipher-arcfour-03.txt, July 1999.

[87] M. Refik, T. Gene, V. Els, and Z. Stefano, *Kryptoknight authentication and key distribution system*, In Proceedings of the 1992 European Symposium on Research in Computer Security - ESORICS 1992, 1992, pp. 1–16.

[88] R. Rivest, *S-expressions*, Internet Draft(Work in Progress), May 1997, http://theory.lcs.mit.edu/ rivest/sexp.txt.

[89] R. L. Rivest, A. Shamir, and L. M. Adelman, *A Method for Obtaining Digital Signatures and Public-Key Cryptosystems*, Tech. Report MIT/LCS/TM-82, MIT, 1977.

[90] P. Samarati and S. De Capitani di Vimercati, *Access control: Policies, models, and mechanisms*, Foundations of Security Analysis and Design (R. Focardi and R. Gorrieri, eds.), LNCS 2171, Springer-Verlag, 2001.

[91] Bruce Schneier, *Applied cryptography: protocols, algorithms, and sourcecode in C*, John Wiley and Sons, New York, 1994.

[92] B. Clifford Neuman J. G. Steiner and J. I. Schiller, *Kerberos: An authentication service for open network systems*, Winter 1988 USENIX Conference (Dallas, TX), 1988, pp. 191–201.

[93] Aura T., *Comparison of graph-search algorithms for authorization verification in delegation networks*, Proceedings of the 3rd Nordic Workshop on Secure IT Systems (NordSec '97), 1997.

[94] W. Tuchman, *Hellman presents no shortcut solutions to des*, IEEE Spectrum **16** (1979), no. 7, 40–41.

[95] B Tung, C Neuman, M Hur, A Medvinsky, S Medvinsky, J Wray, and J Trostle, *Public key cryptography for initial authentication in kerberos*, IETF, June 1999, Internet-Draft draft-ietf-cat-kerberos-pk-init-09.txt.

[96] International Telecommunications Union, *Itut recommendation x.509: The directory: Authentication framework*, Tech. Report X.509, ITU-T, 1997.

[97] PKCS♯5 v2.0: Password−Based Cryptography Standard, *Storing certificates in the domain name system*, Tech. report, RSA Laboratories, March 1999.

[98] Lampson B. W, *Protection*, 5th Princeton Symposium on Information Science and Systems, Princeton, 1971, pp. 437–443.

[99] Joel Weise, *Public key infrastructure overview*, Tech. report, Sun MicroSystems, August 2001.

[100] Martin Abadi Edward Wobber, Michael Burrows, and Butler Lampson, *Authentication in the taos operating system*, Proceedings of the 14th ACM Symposium on Operating System Principles (Systems Research Center SRC, DEC), ACM Press, 1993, pp. 256–269.

[101] ANSI X3.106, *American natiobal standard for information systems - data link encryption*, 1993.

[102] Philip Zimmerman, *The official PGP user's guide*, MIT Press, prz@acm.org, 1994.