Title	Rev	Date
Directory Enabled Networks	A	2000-04-14
Author	Approved	
Henrik Jaakkola		

AU-SYSTEM

Directory Enabled Networks

A Study about Security and Performance

Abstract

This paper is an investigation of the possibility to implement a standard based security solution for the X.25 I/O Mediator using a directory to store security information. The X.25 I/O Mediator is a gateway between a TCP/IP network and an X.25 network enabling clients residing on the TCP/IP side to contact Ericsson AXE input/output Systems, IOGs, residing on an X.25 network. The X.25 I/O Mediator security concept, however, is poor and let clients perform many more operations than desired. This paper suggests one way in how to secure the X.25 I/O Mediator and the IOGs, by denying clients to carry out unwanted operations. The solution is based on the Directory Enabled Networks concept, DEN, which is an initiative to build intelligent networks where information about the network and users is stored in a single directory structure. The information stored is accessible via the Lightweight Directory Access Protocol, LDAP. The X.25 I/O Mediator contacts a Policy Decision Point, PDP, when it needs to find out if a certain operation, issued by a certain client, is allowed. The PDP then makes a decision based on information stored in the directory and returns an answer to the X.25 I/O Mediator which either grants or denies the requested operation. The conclusion is that it presently is not recommendable to use the DEN concept since it is far away from being a completed standard. The idea to use an LDAP directory to store the information about what a user is allowed to do, however, is worth to be investigated further, since the information stored suits a directory very well. It is though advisable for the time being that the proposed solution to store security information in the directory should be in a non-standardised way thus speeding up the time it takes to get the necessary information from the directory.

Contents

Figures

1 Introduction

1.1 Background

A directory is a type of database that is designed, built, and populated with data to allow users to locate objects using information associated with the objects. The structure of a directory makes it very suitable to contain information that is small in size and that does not change very often. The directory-enabled networking concept is one of the most topical issues in the computer branch these days. There is a new way of thinking in network management. It is desirable to have one single directory structure as a centralised repository where information about everything in the network is accessible via the widely adopted access protocol Lightweight Directory Access Protocol, LDAP.

The computer consulting company AU-System is interested in this new concept and the possibilities to use it. The target goal is to use the concept in security management of the X.25 I/O Mediator. X.25 I/O Mediator is a gateway between a TCP/IP network and an X.25 network enabling IP clients to access Ericsson AXE Input/Output systems, also called IOGs residing on the X.25 network side. The X.25 I/O Mediator allows a client to use telnet or ftp to communicate with IOGs and also lets files and alarms to be sent over from the IOGs to predefined computers on the IP network side. The current X.25 I/O Mediator security concept is however very insufficient and therefore AU-System has decided to make an investigation in how an improved security concept can be used in relation with the new directory-enabled network concept.

1.2 Goal

One goal of this thesis project was to describe how directories and LDAP work and to describe the differences between a directory and a relational database. Furthermore, the project is about the current standardisation work going on how to define a single directory structure for a company. The work will also show how the X.25 I/O Mediator, together with security information, could be defined in such a directory. Also, it will be described what different attributes such a security concept could be based on and what is interesting to restrict on the X.25 I/O Mediator.

1.3 Outline

The paper starts with a theoretical part describing LDAP directories, Section 2. Section 3 presents a comparison between directories and relational databases. After that, in Section 4, the current standardisation work and how it is possible to store policy information in a directory is investigated. The policy-based networking concept is shown to illustrate how policy information could be used. In the same section the Directory Enabled Networks, DEN, which is an initiative to build standardised intelligent networks with a directory as a centralised

repository is briefly described. A look is also taken at the IETF's policy information LDAP core schema, which is one way to store policy information in a directory. Section 5 deals with the Ericsson AXE I/O System and in Section 6 the X.25 I/O Mediator concept is presented. After that, in Section 7, a look at the security problem in the X.25 I/O Mediator is taken and in Section 8 a directory-based solution of the problem is proposed. The thesis ends up with a conclusion showing the current state of standardisation of DEN and the policy structure and also if it is a good idea to use the DEN concept on the X.25 I/O Mediator at the time being.

2 The LDAP Directory

2.1 History

A directory is merely a type of database that is distinguished by its purpose: it is designed, built, and populated with data to allow users to locate objects using information associated with the objects.

The use of directories has increased rapidly over the last decade and there is no doubt that a directory is an enormous help, concerning storage of information about users and applications. The problem however, is that there are many different directories localised in different places in the network and many of these store the same information. This makes it difficult to update and preserve control of the directories.

The first attempt to invent a directory that was not application-specific or designed for one specific operating system was the X.500. It was defined by the OSI/ITTF, published in 1988 and updated in 1993. X.500 is a series of standards that specifies how information can be stored and retrieved in a global directory service structure. X.500 is though a very complex structure and it has never been truly adopted by developers.

X.500 is also based on the OSI network protocols. This fact made it even more difficult to adopt the X.500 standard since the development of network technology has shown that the TCP/IP suite with its simplicity, speed and low cost has been chosen by developers as the protocol suite to use in the networks today. X.500 was never designed to work in TCP/IP networks.

The X.500 uses a protocol called Directory Access Protocol, DAP, as the protocol to access the directory. DAP is very complex and difficult to implement and most of the implementations perform badly. The two first attempts to escape DAP were the protocols Directory Assistance Service, DAS, and Directory Interface to X.500 Implemented Efficiently, DIXIE. Clients would connect to a gateway server speaking DIXIE or DAS and the gateway would translate these TCP/IP requests into X.500 requests. These two protocols, with DIXIE as the favourite, were very popular among implementers and it was obvious that DAP was not the protocol to use. However both DIXIE and DAS were tied closely to a single X.500 implementation called Quipu. An attempt to standardise the Dixie/DAS to work with all versions of X.500 was specified by the IETF´s OSI Directory Service Working Group in 1992 and the result was Lightweight Directory Access Protocol, LDAP.

The developers of LDAP simplified the DAP protocol in four important areas [1]:

- Functionality. LDAP simplifies the implementation of clients and servers and still preserves most of DAP´s functionality.

- Data representation. In LDAP most of the data is carried as simple text strings, although messages are still encoded in binary strings. This increases the performance and simplifies implementation.
- Encoding. LDAP only uses a subset of the encoding rules in X.500.
- Transport. LDAP runs directly over TCP and does not require the multilayered OSI networking stack.

This protocol was widely accepted and used as a front end for X.500 based directory services. Since that time LDAP has grown to be something beyond just an access protocol. LDAP was used in the beginning as DIXIE and DAS were, as a translator to connect to X.500 directories. But when developers noticed that almost every request to an X.500 directory was made with the use of the LDAP to DAP translator the idea to get rid of the X.500 directory was introduced. The first standalone LDAP server was implemented and released in early 1995 at the University of Michigan. It was called “slapd”, which stands for standalone LDAP daemon. LDAP still remains quite compatible with X.500 but supports several issues which X.500 does not, e.g. confidentiality discussed in Section 2.2.5.

2.2 Specification

The current version of LDAP is LDAPv3[3]. It consists of 8 features, which together provide a solid and efficient way of organising, retrieving and maintaining a complete and efficient directory structure.

2.2.1 The Information Model

The Information Model defines the structure of how the data is maintained and referenced in the directory. The directory consists of entries. An entry is an object, which holds information about that object. The entry consists of attributes with values. An example of an entry is shown in Figure 1.

cn:	Henrik Jaakkola hja
sn:	Jaakkola
telephoneNumber:	+46 (0)8 7267582
uid:	Hja
mail:	Henrik.Jaakkola@ausys.se
employeeNumber:	924

Figure 1. Example of an entry with attributes and values.

As seen in Figure 1 attributes can have more than one value. The ‘cn’ attribute for example has the values ‘Henrik Jaakkola’ and ‘hja’. Each attribute in the entry has a type and one or more values. Each attribute also has an assigned syntax that specifies what the value of the attribute is decoded in. The ‘cn’ attribute is standardised and defined in [2]. In Section 2.2.3 it is described how to define attributes and objects.

2.2.2 The Naming Model

The directory is organised hierarchically in a tree structure with an entry defined as the root of the tree. This entry, the root, is typically an organisation, e.g. AU-System.Every entry has a distinguished name, DN, that pinpoints that specific entry in the directory. The DN consists of relative distinguished names, RDNs, which are separated with commas. Remember Figure 1 of the entry in the previous section. In the directory it could be placed as shown in the Figure 2. The DN of the entry would be ‘cn=hja, ou=users, ou=Stockhlm, dc=ausys, dc=se’.

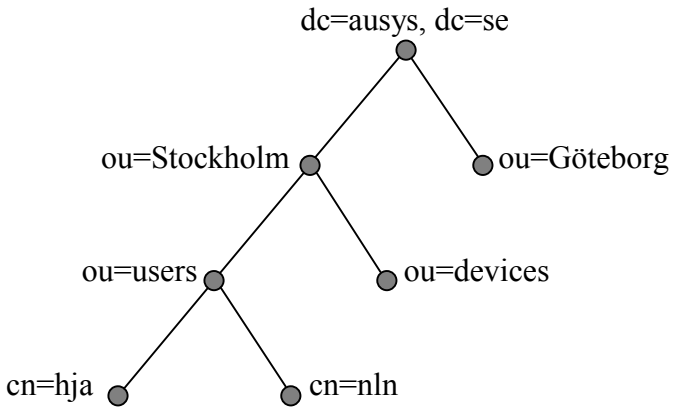

Figure 2. The naming model.

2.2.3 The Schema

Every entry in the directory has a special attribute called objectclass. The value of this attribute defines the class of the object. The LDAP Schema defines what type of attributes that must and may exist in an object of a certain class.

Every class, except a class called `top`, is derived from another class, thus it is possible to define a new class that inherits properties from another class.

Taking another example, let us again look at the entry in Figure 1. That entry is defined to be of the class `inetOrgPerson`. `InetOrgPerson` is derived from a class `organizationalPerson`, derived from a class `person` which is derived directly from the `top` class. The structure used to define classes can be seen below (in this case the object class `Person` defined in [2]):

```
(2.5.6.6 NAME 'person' DESC 'Standard Person Object Class'
        SUP 'top'
```

```
STRUCTURAL
MUST( sn $ cn )
MAY(userPassword $ telephoneNumber $ seeAlso $ description))
```

Above we can see that the class `Person` inherits from the class top. This is defined after the SUP which stands for super. The `top` class is the root class, where all classes originally derive from. The `top` class is abstract, see below for description, and defines that all classes must contain the attribute 'objectclass'.

We also see that the object class `Person` is a structural class. This is the default and normally not shown in the definition. It means that the object class `Person` can be instantiated in the directory. The two other types of object classes that exist are the types abstract and auxiliary. Abstract means that they can not be instantiated, but only inherited from. Auxiliary classes are classes that can be added to an existing class, thus giving it new attributes and values without sub classing the existing class.

Instances of the class `Person` must contain attributes of the types 'sn' and 'cn'. Further more instances may contain attributes of the types called 'userPassword', 'telephoneNumber', 'seeAlso' and 'description'. Attributes must also be defined in the schema. The attribute definition of 'cn' is:

```
(2.5.4.3 NAME 'cn' DESC 'CommonName Standard Attribute'
        SYNTAX 1.3.6.1.4.1.1466.115.121.1.15
```

The SYNTAX variable defines the syntax of 'cn'. The syntax of 'cn' in human readable form is "Directory String" (A string in this syntax is encoded in the UTF-8 form of ISO 10646). In the definition the syntax is displayed as a number which is called OID, Object IDentifier. The beginning of the definitions in the examples above also starts with an OID (2.5.6.6 and 2.5.4.3). The usage of OID is to remain compatible with X.500. LDAP favours the string that follows the NAME in the definition above, i.e. in these cases 'person', 'cn' and 'directoryString'. This means that when designing an LDAP directory you should avoid the OID and use the string representation instead.

2.2.4 The Functional Model

The Functional Model defines how you actually can access the data in the directory. There are nine different operations defined in the LDAPv3 RFC [3]:

- **Abandon**. Allows a client to request that the server abandon an operation in progress.
- **Add**. Allows a client to request the addition of an entry.
- **Bind**. Enables the exchange of authentication information between LDAP server and client.
- **Compare**. Allows a client to compare an assertion provided with an entry.

- **Delete**. Allows a client to request the removal of an entry.
- **Modify**. Allows a client to request a modification of an entry.
- **Modify DN**. Allows a client to change the leftmost, least significant, component of the name of an entry.
- **Search**. Allows a client to request a search to be performed in a single entry, from entries immediately below a particular entry, or from a whole subtree of entries.
- **Unbind**. Disables the exchange of authentication information between LDAP server and client.

2.2.5 The Security Model

Security is of course an important issue, since information stored in the directory may be more than just white paper information. There are two important issues when discussing the security. The first is to restrict who has access to what in the directory. The second is how to secure the data being sent between a client and a server.

The current version of LDAP does not define a common access control model. This means that the directory services that exist on the market today have different ways to manage this security. Standardisation work is though going on, and currently the IETFs LDAP Extension Working Group[8] has an Internet draft describing a standard access control model.

When an LDAP client wants to speak to an LDAP server it opens a connection to this server. At some time it can authenticate to the server to enjoy privileges depending on what it authenticates as. The process of authenticating is the operation “bind”, see the previous section.

There are different ways for a client to authenticate to the server. Simple authentication means that the client connects to a server using the “bind” operation and binds to a specified DN sending the password in unencrypted form over the network. The server then checks the entry corresponding to the DN and compares the password sent with the value of the attribute ‘userpassword’ in the entry. If the password matches the client is authenticated. If it does not, the authentication operation fails and an error message is sent back to the client. There are also ways to send password in encrypted form using for example secure sockets layer, SSL. SSL is based on public key cryptography and can provide very high security. It includes strong authentication, signing and encryption services. LDAPv3 also defines how to secure communications via the Simple Authentication and Security Layer mechanism, SASL. SASL enables support for authentication, encryption and signing services. Although it provides no security itself, it allows application protocols such as LDAP to negotiate security parameters[1].

2.2.6 The LDAP Data Interchange Format

The LDAP Data Interchange Format, LDIF, is a standard text-based format for describing entries in a directory. This enables the data in the database to be exported from and to the database in an easy manner. An example of the text representation is shown below. The entry corresponds to an `inetOrgPerson` that inherits from `organizationalPerson` and `Person`.

```
dn: cn=hja, ou=users, ou=Stockholm, dc=ausys, dc=se
objectclass: person
objectclass: organizationalPerson
objectclass: inetOrgPerson
objectclass: top
sn: Jaakkola
cn: Henrik Jakkola
cn: hja
telephoneNumber: +46 (0)8 7267582
uid: hja
mail: Henrik.Jaakkola@ausys.se
employeeNumber: 924
```

Entries like this can be read by the LDAP server and stored in the database. Entries in a file are separated by a blank line and if a line starts with a space it means “continue from the line above”. It is also possible to change or remove existing entries stored in the database via the LDIF format. For Example changing the attribute and value ‘cn: Henrik Jakkola’ to, the correct name, ‘cn: Henrik Jaakkola’ would look like:

```
dn: cn=hja, ou=users, ou=Stockholm, dc=ausy, dc=se
changetype: modify
replace: cn
cn: Henrik Jakkola
cn: Henrik Jaakkola
```

This changes the specified ‘cn: Henrik Jakkola’ and not the ‘cn: hja’. Other defined changetype values are add and delete.

2.2.7 LDAP Protocol

LDAP uses TCP as the transport protocol, and the LDAP protocol defines how the requests from clients and the answers from servers are carried over the wire, i.e the format of the messages sent between the client and the server. For example in the protocol it stipulates that every request is carried in a common message format. Another example is that entries, which are returned from the server on search request from a client are carried in separate messages, thus allowing the streaming of large result sets[4]. Readers that are interested in more information on the LDAP Protocol are referred to[3].

2.2.8 The LDAP API

The creators of LDAP early developed a standard API to facilitate the development of directory-enabled applications. The original API was developed at the University of Michigan and included a C programming Library. Today there exist APIs for several different programming languages for example C++, Java and Perl.

3 Directories versus Relational Databases

3.1 Introduction

This section presents an overview of the characteristics of directories compared to general relational databases. There are key differences to consider when it is time to choose whether one wants to use a directory or a relational database. This section concludes with a summary of applications suitable for a directory but not for a relational database and vice versa.

3.2 Characteristics

3.2.1 Read and write

The main difference between a directory compared to a relational database is that the directory is designed to be read from far more often than written to. The relational database on the other hand is designed to handle read as well as write operations. This means that a directory is faster to be read from than a relational database, but on the other hand slower concerning writing information to it.

3.2.2 The data contained

The directories are suitable for storing relatively small pieces of information such as information about a company, users of that company, and configuration files for a web server. On the contrary relational databases are designed to also handle big files. Moreover the directories support a relatively simple transaction model that involves only a single operation and a single directory entry. Relational databases are able to handle large and diverse transactions, spanning multiple data items and many operations.

3.2.3 Distribution and Replication

Another important difference is that a directory is more suitable to have the data distributed among several servers. It is also possible to find databases that allow limited distribution of data, but the scale of the distribution is quite different. The typical relational database allows storage of one table at one place and another table at another place. The ability to make queries involving both of these sites exists, but the performance is often a problem[1]. Closely related to distribution is replication. Replication means that data is copied, shadowed, from one location to another. This makes it possible to maintain reliability and availability. If one server breaks down another can handle the request. The directories handle replication better than relational databases. The reason is that data in databases must be in strongly synchronisation all the times. In the directory however a small difference in synchronisation is usually acceptable[1].

3.2.4 Summary

As a summary we take some applications that are suitable for a directory but not for a relational database and vice versa.

An application that needs *information linking* is suitable for a database but not for a directory. Examples of such applications are Accounting Systems and Enterprise Resource Planning.

Whenever an application needs to be *distributed* the choice should be a directory and not a relational database. Examples of such applications are address book support for mail clients, message system configuration and support of X.509 based Public Key Infrastructure.

Some problems could be solved with a relational database as well as with a directory. A good example is white pages such as telephone books. They work well in a directory and can also be stored in a relational database.

4 Policies and Directories

4.1 Introduction

Managing a network is a complex task nowadays. Especially concerning security aspects. Security solutions that exist today are often complicated, poor in scalability and there is a big need to find a way to simplify the managing of security. The idea to let directories contain information about the network, hosts and users is not a new idea. What is new is the idea to have one single directory from where all information in the network is available. This directory can also store information about what a user, or a computer, is allowed to do and thus making the network structure much easier to manage and maintain.

The ultimate goal of a directory-enabled network is to establish a single directory structure that is accessible to the entire enterprise[5].

4.2 Policy-based Networking

A directory itself is only an information container and the access protocol, LDAP, only supports retrieval and update operations toward the directory. The DEN, specification, see Section 4.3, defines the class Policy as a class that encapsulates information governing the use of and interaction between network resources and services in a particular context. A policy is a named object that represents an aggregation of policy objects.

However, these characteristics raise a number of questions:

- How and where should the actual access control functionality, i.e. the decision on whether the requested operation should be allowed or not, be implemented? The directory itself does not promote this type of functionality.
- What about performance? Can the directory support the required number of inquiries rate?
- Will the solution be robust enough? What if the directory itself, or the connection to it, fails?

These issues are also discussed by the DEN committee work. Ideas on policy enforcement and reliability have resulted in a proposed implementation framework discussed below. Note that the DEN policy enforcement addresses multiple policies. The initial focus has been on routing and traffic management, but security management is also recognised as a policy area.

Policy-based networking is complex and requires more than just a directory supporting retrieval functionality. The DEN committee has proposed a concept on policy enforcement in the network.

The roles required in policy enforcement include: information storage, policy requester, policy interpreter and policy enforcer. The concept is shown below in Figure 3.

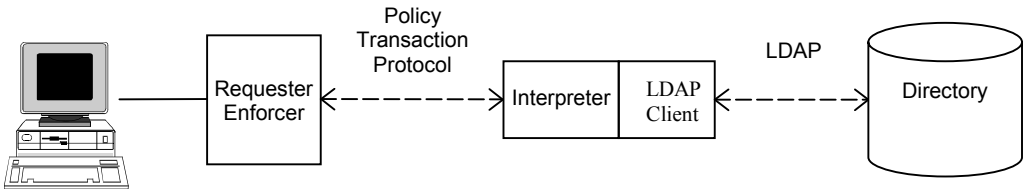

Figure 3. Policy-based networking.

4.2.1 Policy Requester

A policy request is triggered when a policy client, typically a router, observes a demand to access network resources. The trigger, for example, could be the launch of an application. The ability to access that application is determined by permissions contained in a given users profile and the enforcer, the router, submits a request to a policy server, asking it to make a decision. Ideally, this request is transmitted to the policy server via a policy transaction protocol, see Section 4.2.5. The policies can of course be retrieved and cached locally on the enforcing device, depending on the request type.

4.2.2 Policy Interpreter

Policy interpretation is the role performed by the network device that manages state and evaluates the particulars of the resource request[5]. The interpreter is the actual decision-maker, and may be required to consult a variety of different information before it can make a policy decision. For example the request may depend on how much load there is on the network, what time of the day it is etc.

4.2.3 Information storage

The information storage is the LDAP directory containing information about policies and profiles. This is where the Interpreter collects the information using the LDAP protocol.

4.2.4 Policy Enforcer

Enforcement is the role performed by a network node to ensure that the policy lease is realised. If the Enforcer is a router and the request was to access an application the router either forwards the request or denies forwarding depending on the policy returned by the Interpreter.

4.2.5 Policy transaction protocol

The policy transaction protocol used by the enforcement device is a detail currently being discussed by IETF working groups. There are two different protocols in question. One of the proposed standards is the Common Open Policy Service protocol, COPS, and the other is using an extended version of SNMP.

4.3 Directory Enabled Networks

Directory Enabled Networks, DEN, is an initiative to build intelligent networks. DEN, which is currently being developed by the Distributed Management Task Force, DMTF, uses a directory to store the information. The structure is based on X.500 and the Common Information Model, CIM and will use LDAP to communicate with the directory. DEN was initiated in 1997 by Microsoft and Cisco, and today there are a lot of companies supporting the development of DEN.

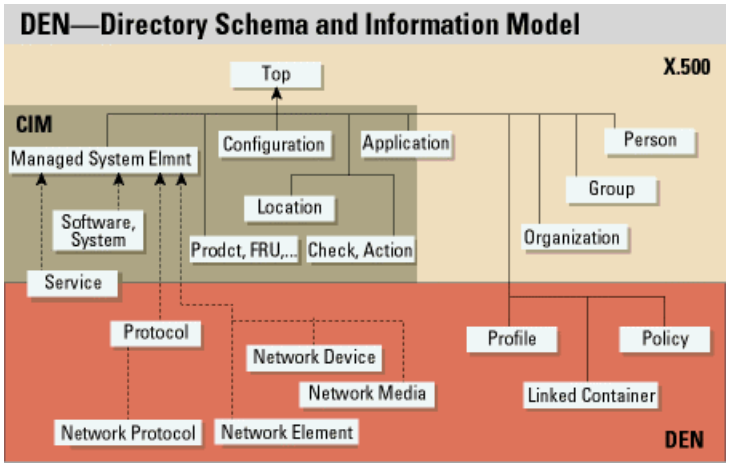

Figure 4. DEN Base Schema and Information Model.

Figure 4 above shows the base classes specified in DEN. A future usage of DEN in the network will probably simplify many hard tasks and facilitate security management in the network.

Unfortunately, the standardisation of DEN seems to have stalled a bit lately. There are many steps still to take before it is a completed standard. A big question currently in the standardisation is what protocol the Policy interpreter should use to communicate with the policy requester and enforcer, see Section 4.2.5. There is also still work left to be done on the standardisation of the specific DEN classes and how they should be stored in the directory.

4.4 The Policy Framework Working Group

The IETF is currently also defining a standard for policy-based networking. This is done in the Policy Framework Working Group. This group's work is related to the DMTF's work and is kept as close to the DEN/CIM policy model as possible. The group has developed an internet draft describing how policies may be stored

in a directory. The work in the Policy Framework Working Group is presently focused on policies for quality of service (QoS), but the Qos model is derived from a general policy information model, also developed by this group, see Figure 5 below. That model is truly general and can be extended to implement various different types of policy structures where one example would be security management.

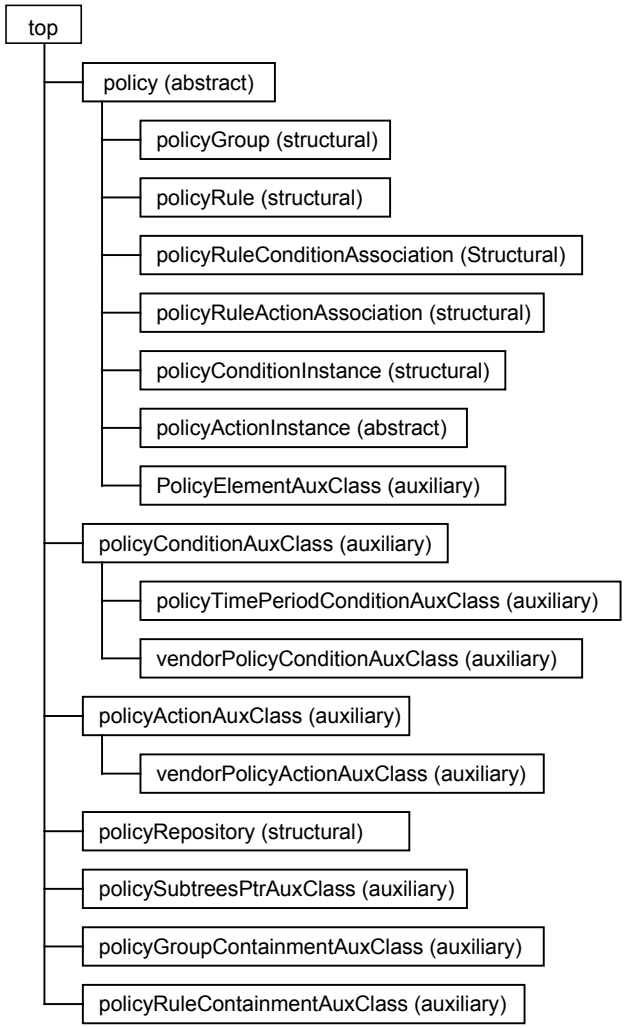

Figure 5. General Policy Model Class Hierarchy.

Figure 5 above shows the class hierarchy proposed by the Policy Framework working group. Further information about the classes is available in Section 8, at [6] and in [7].

5 The AXE Input/Output System

This section gives a short overview of the SP-based IO System in AXE, which is the system that users can connect to using the X.25 I/O Mediator described in next section.

SP is an abbreviation for Support Processor, the separate processor that controls the IO system. There are several SP-based IO systems that exist in AXE today, such as IOG11A, IOG11B, IOG20B and IOG20C. These systems are rather similar and since this is an overview we will just refer to it as IOG which stands for Input Output Group.

The AXE IO system can generally be described as follows [9]:

Handling of data
Handling of data to and from the Central Processor, CP. This means that the IOG is the IO interface to the world outside an AXE exchange. The data may be alphanumeric, like printouts, commands and alarms, statistic and charging data. The data may also be binary like software backups, etc.
Secondary Storage
A secondary storage of information on magnetic media, e.g. hard disk, optical disk or flexible disk.

The hardware of an IOG, here a simplified standard IOG20, can be seen in Figure 6 below.

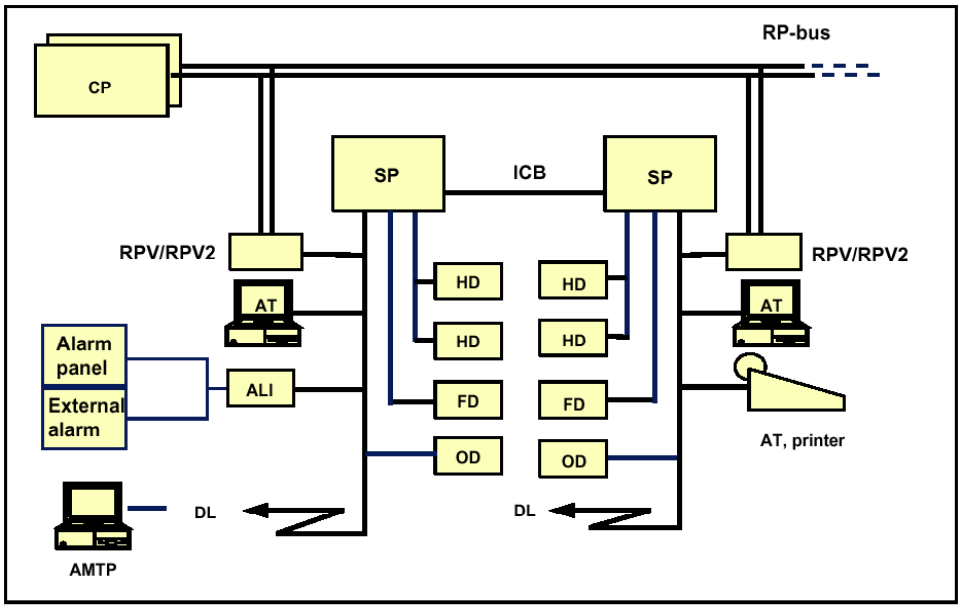

Figure 6. The hardware of an IOG.

As seen in the Figure 6 above there are, in the standard IOG concept, two identical SPs which each control a number of IO devices. This is done as a precaution against faults arising in one of the SPs. The IO devices in Figure 6 are:

- AT - Alphanumeric Terminal
- AMTP - Alphanumeric terminal over Ericsson MTP
- ALI - Alarm Interface
- HD - Hard disk
- FD - Flexible Disk
- DL - Data Link
- OD - Optical Disk

An IOG as described Figure 6 with two SP's each controlling a number of IO devices is called a Support Processor Group, SPG. An SPG and an IOG are equivalent concepts. An SPG may only have two SP's.

The Alphanumerical Terminal, AT, is the device used for man to machine communication. The ATs are thus used for sending commands and receiving printouts. An AT can be any type of asynchronous terminal, normally a personal computer, PC, or a Work Station, WS. It can also be a line printer such as the alarm printer in Figure 6. The AT may also be used for machine to machine communication. There are a number of different communication programs for PCs and WSs. Examples are FIOL for DOS, WINFIOL and AXEUSE for Windows, or TENUX via WIOL for UNIX.

The data communication for operations and maintenance in the AXE exchange is provided via a concept called Data Communications Subsystem, DCS. The structure of DCS is based on the OSI reference model. The only IOG that supports the IP protocol at the network level, i.e. level 3 in the OSI reference model, is the IOG20. Previous IOGs use X.25 at the network level. This is where the X.25 I/O Mediator, described in the next section, will help. The X.25 I/O Mediator makes it possible for applications residing on an IP network to communicate with IOGs that only support X.25 network communication.

6 The X.25 I/O Mediator

6.1 Background

AU-System has been assigned to develop a solution for X.25 communication support for EMRK applications. The solution for this communication is called the X.25 I/O Mediator.

6.2 The X.25 I/O Mediator Concept

The X.25 I/O Mediator concept is based on hiding the X.25 network from the EMRK application, which will instead use TCP/IP (TELNET and FTP) and connect to the X.25 I/O Mediator when accessing the X.25 network. This means that the X.25 I/O Mediator acts as a limited gateway between the X.25 network and the TCP/IP network. The EMRK application does not have to configure X.25 parameters, or even be aware of that it is using X.25 in order to access the IOG. A schematic figure of the X.25 I/O Mediator Concept is shown below.

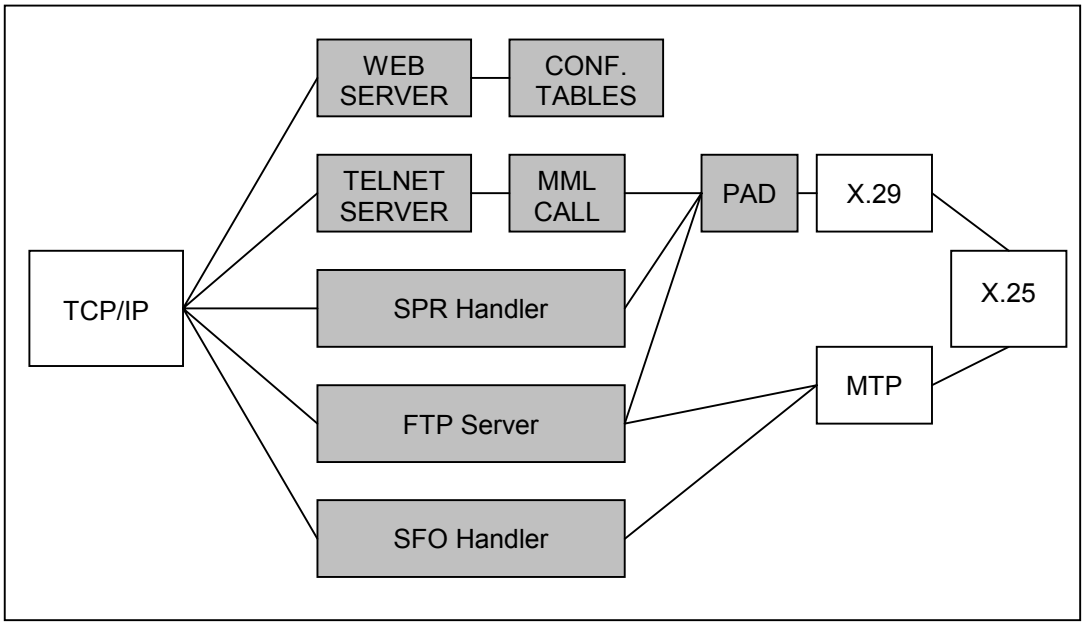

Figure 7. X.25 I/O Mediator Concept.

As shown in Figure 7 there is one Telnet server and one FTP server in the Mediator which clients can use to connect to the Mediator and communicate through to operate on IOGs on the X.25 network side on the mediator. Information on the tables that contain data about the different IOGs and ATs available can be received from the web server. The web server also makes it possible to configure the tables. The other module are the SPR Handler which is used to handle spontaneous printouts from the IOG and the SFO Handler which is used to handle the spontaneous files an IOG can send at certain times.

7 The security problem

7.1 Introduction

The X.25 I/O Mediator concept is extremely limited concerning security. The only security feature supported in the X.25 I/O Mediator is the procedure when a user logs on the X.25 I/0 Mediator. Here authorisation is gained by checking the users NT-password. This means that registered users on the NT-server are able to log on to the X.25 I/O Mediator. When logged on however, users have privileges to do operations far more than always wanted.

When a user is connected via telnet, see Figure 8, he can choose between four different operations: ‘cmd’, ‘help’, ‘MMLCall IOG AT’ and ‘exit’.

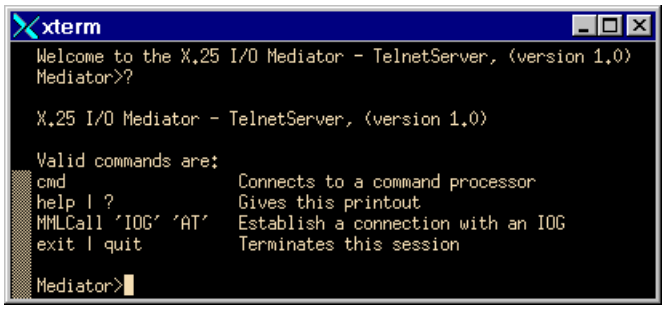

Figure 8. X.25 I/O Mediator Telnet Session.

If he issues the command ‘cmd’ the user is connected to a command processor on the X.25 I/O Mediator. The X.25 I/O Mediator is running on a Windows operating system which means that the ‘cmd’ command will basically start up a DOS prompt where the user can type in commands. In this session the user is allowed to do critical operations such as start and stop the different services provided by the X.25 I/O Mediator. The other available command, of interest, is the command ‘mmlcall IOG AT’. This command starts up an AT-session to the specified IOG and AT where the user can type MML commands. The security is now on the IOG side since the user also has to log on to the IOG. This means that the security here is adequate since the user has given privileges to the specified IOG.

When the user connects to the Mediator using FTP it is also important that the user can be restricted before he tries to get or put files on the defined IOGs. When the user is logged on via FTP the X.25 I/O Mediator is already configured with information on how the communication with the IOG should be done. This means that the user does not need to log on the IOG. It is the X.25 I/O Mediator that handles the communication with the IOG without any more required information from the user. Therefore using the X.25 I/O Mediator via FTP a user on the IP side can store and take away files as he wishes. All that is needed is that the user is registered in the NT-server and that the IOG he wants to store or take away files from is configured for the Mediator.

7.2 What to restrict

Understanding that implementing security in the Mediator is needed, the question arises of what should be restricted. It is possible to implement a very complex security system but the question is if it is necessary.

One example is to implement restrictions that when a user connects to the Mediator via Telnet and issues the command ‘cmd’ we restrict what users are allowed to do this, since that user has privileges to do almost everything on the X.25 I/O Mediator. Another example is when the user issues the command “MMLCall IOG AT” restrictions are implemented that check that the user is allowed to contact the specified IOG and AT. Moreover it can be implemented what time of the day the user is allowed to do this. When this command is issued and the AT-session starts, there are a great number of commands the user can type in. It is estimated that there are about 5000 commands available. The question is if it is a reasonable idea to also implement restrictions on what commands the user can type in during the session. This might be too complex since the number of different commands is so great. Assuming it is motivated, a time condition might be considered as well, i.e. a restriction about the time of the day the user is allowed to perform the different commands.

FTP sessions are a little bit different. When connected to the Mediator via an FTP session the user never types any specific MML commands. It is rather the X.25 I/O Mediator that translates the FTP commands to understandable MML commands for the IOG. The user that is connected to the X.25 I/O Mediator can issue the following FTP commands DIR, LS, CD, PUT, GET, BYE, QUIT, and HELP.
In the FTP case an example of restriction could be to restrict a user from IOGs he is not authorised to connect to. Moreover, it might be a good idea to implement restrictions that a user might retrieve files but not store them and vice versa. The time condition, as in the Telnet session, might be considered too.

8 The Solution

8.1 Introduction

8.1.1 Solution approach

There are many different ways to implement security restrictions in the Mediator. The solution proposed in this paper involves a Directory and is based on the policy work currently going on in the IETF Policy Framework working group, see Section 4.4.

The original idea was to base the solution entirely on the DEN concept, currently being developed by the Distributed Management Task Force, DMTF, see Section 4.3. But after having investigated the possibilities of using the DEN concept the conclusion reached was that it would afford too many own inventions.

Therefore the work is based on a draft made by the IETF's Policy Framework working group called Policy Framework LDAP Core Schema. That draft proposes how it is possible to store policy information in a directory. The group has tried to base its work on the CIM/DEN model so the solution proposed in this paper is close to DEN anyway. The reason to choose the IETF approach is that there exists a draft on how the information can be stored in the directory. It is a very general model and has proven possible to be extended with security-related policies needed for the X.25 I/O Mediator.

To make the solution more complete the X.25 I/O Mediator, with its table, is also defined in the directory. This makes it possible for future versions of the X.25 I/O Mediator to be truly incorporated in a directory-enabled network.

8.1.2 The basic structure

The basic structure of how to solve the problem defined in the previous section is shown in the Figure 9.

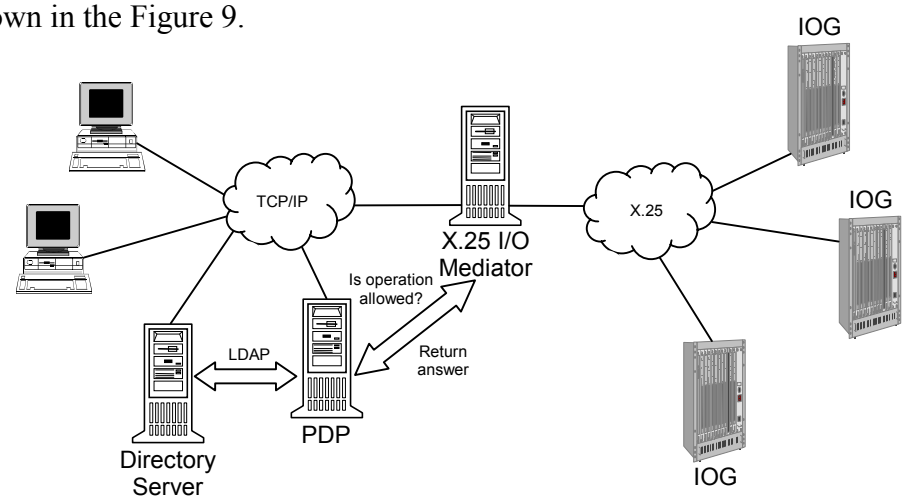

Figure 9. The basic solution.

When the Mediator needs to find out if a user is allowed to do an operation it is designed to send a question to a Policy Decision Point, PDP. The question contains information about the user and what he is requesting to do.

The PDP then tries to find a policyrule that matches the given conditions in the request. All of the different policyrules are contained in a Directory, under which the PDP can connect to and download from. The PDP then makes a decision based on the policyrules and finally returns an answer to the Mediator.

8.2 What to restrict

The choice of authorisation this paper proposes is based on the architecture of IOGs, the design of the X.25 I/O Mediator and common sense. It is suggested that security restrictions should, at a first step, be implemented as specified in the following two sections.

8.2.1 Telnet

When a user is connected via Telnet he has six available commands to issue. Four of them are not interested basing restrictions on. They are to obtain help information: ‘help’ and ‘?’, and to log of the X.25 I/O Mediator: ‘exit’ or ‘quit’. There is no need to restrict these commands since they only give the user information and make it available for him to log off.

The critical commands are the two commands ‘cmd’ and ‘mmlcall IOG AT’.

When the user issues the ‘cmd’ command it is of the most importance that this only is available to users who have administrative privileges. This command gives the user access to the whole file system on the X.25 I/O Mediator. The telnet session behaves as a normal DOS-prompt for the user. Therefore a user that has issued this command must be the administrator of the X.25 I/O Mediator. The arguments we can base this decision on are the following:

- Telnet
- user
- cmd

It means that the question sent to the PDP could generally look like the function call isAllowed(Telnet, User, Command) where the first argument is the type of service the user requests, the second the ID of the user and the third the command the user wants to issue, i.e. ‘cmd’. We then let the PDP decide based on these arguments if the user is allowed to issue the command ‘cmd’. The PDP then returns a Boolean answer, TRUE or FALSE, to the X.25 I/O Mediator, which either grants or denies the user access.

When the user issues the command “MMLCall IOG AT” the X.25 I/O Mediator should contact the PDP and ask it to decide if the user is allowed to contact the specified IOG and AT. Hence we have the following arguments to base the decision on:

- user
- MMLCall
- IOG
- AT

This means that the question sent to the PDP could be formed like the function call isAllowed(Telnet, User, IOG, AT) and then let the PDP decide based on these arguments if the user is allowed to contact the specified IOG and AT. The PDP then returns a Boolean answer, TRUE or FALSE, to the X.25 I/O Mediator, which either grants or denies the user access.

The general advise is that it is enough to make restrictions based on these attributes on the telnet part of the X.25 I/O Mediator. There is also a possibility to have a time condition in the proposed solution. This time condition makes it possible to generate intervals, for example that a user is only allowed to contact the specified IOG and AT during weekdays but not weekends.

The motivation to not restrict the Telnet part more is complexity of IOGs and the design of the X.25 I/O Mediator. Constructing an authorisation system more complex would be too difficult.

Implementing further restrictions about what a user might do when he has issued the command ‘cmd’ is not necessary since that user should only be the administrator of the X.25 I/O Mediator and he should be able to do all he needs.

Implementing further restrictions when a user has issued the command “mmlcall IOG AT” is also very complex. The X.25 I/O Mediator then starts up an AT session, which allows mml commands to be sent to the IOG. Making an authorisation system on all the different available command is too complex. When the user has issued the command he also has to identify himself to the IOG through the AT session as well, i.e. log on to the IOG. This alone may be enough restriction, but implementing the above limits the user occupying an X.25 channel he is not allowed to use and also makes it possible to have a time conditions.

8.2.2 FTP

When a user is connected via FTP to the X.25 I/O Mediator it is a good reason, as well as a possibility, to do more restrictions than in the Telnet case.

First of all when a user is connected, being placed at the root of the hierarchy, and issues the command CD, Change Working Directory, to get inside an IOG, the X.25 I/O Mediator should be designed to contact the PDP to get approval. The X.25 I/O Mediator should send information about the identity of the user and the name of the IOG that he wants to connect to. The X.25 I/O Mediator should also send a string representing the service and the command the user wants to issue. The Mediator is configured with which AT (i.e. NTN) it should use to contact this IOG with. Sending this variable is therefore not necessary, since the user cannot

choose what AT to contact. Hence we have the following attributes to base the decision on:

- ftp
- User
- IOG
- CD

This means that the question sent to the PDP could be formultaed like a function call isAllowed(ftp, User, IOG, CD). The PDP then decides, based on this information, whether the user is allowed to access the IOG. The decision is returned, as a Boolean answer TRUE of FALSE, back to the X.25 I/O Mediator, which either grants or denies the user access.

It might also be of interest that a user can read from the IOG but not write to it. This is possible since the command the user sends is included in the question. When the user wants to get a file he issues the command "get 'file'" the name on the IOG he wants to store the file on and the command 'get' should be sent. It is also possible to define a more complex access system, such as where the user is allowed to write files. Replacing the variable IOG with the absolute path to where he wants to store the file is one way of doing this. That is not considered anymore in this solution but should be easy to implement if one wants to.

The reason to also send the service ftp is to make it easier for the PDP to distinguish between the two services ftp and telnet/mmlcall.

The time condition could be interesting as well, and there is, as in the Telnet case, a possibility to define time conditions that restrict users to access a certain IOG at a certain time.

8.3 Defining the X.25 I/O Mediator in the Directory

In a directory-enabled network all different nodes in the network are contained in the directory. It would be nice to define the X.25 I/O Mediator in the directory as well. Information contained about X.25 I/O Mediator can be a lot, such as the hostname, install date, running OS etc. What we really want though, excluding the policy architecture, is information in the table, which maps between the IP network and X.25 network, showing available IOGs and ATs etc.

Defining the X.25 I/O Mediator with the DEN specification in mind is not a simple as it sounds. DEN specifies a class `NetworkElement` for nodes in the network. It is inherited from the CIM class `UnitaryComputerSystem` as shown to the left in Figure 10 below. This is where the X.25 I/O Mediator should be placed if the current DEN specification should be followed.

As seen in Figure 10 below there are a lot of classes that each defines new concepts of how to describe the X.25 I/O Mediator. In this solution we simplify the structure and let the X25IOMediator class inherit directly from the `LogicalElement` as shown in the right in Figure 10.

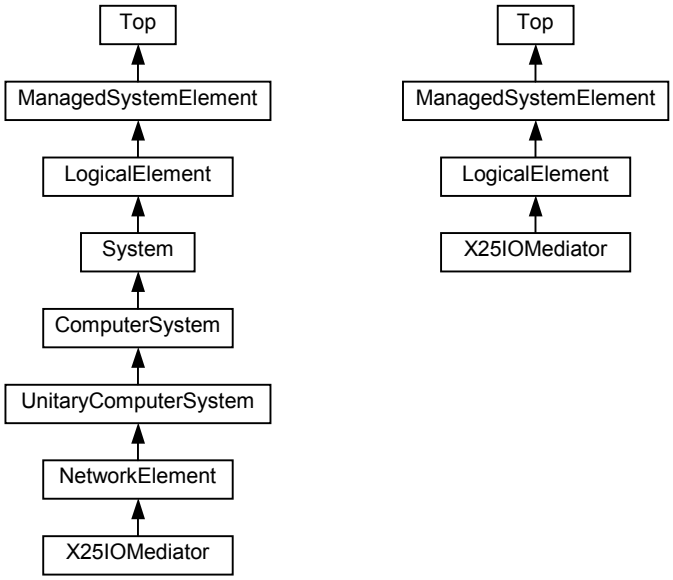

Figure 10. The Class hierarchy.

The class definition is the following:

```
(<OID> NAME 'X25IOMediator' DESC 'An X.25 I/O Mediator'
        SUP 'LogicalElement'
        STRUCTURAL
        MUST()
        MAY(policyDecisionPoint))
```

The attribute ‘policyDecisionPoint’ is a DN pointer to the PDP that also is defined in the directory. From that pointer the X.25 I/O Mediator can get the necessary information it needs to contact the PDP. Figure 11 below shows where the X.25 I/O Mediator can be placed in a company’s directory.

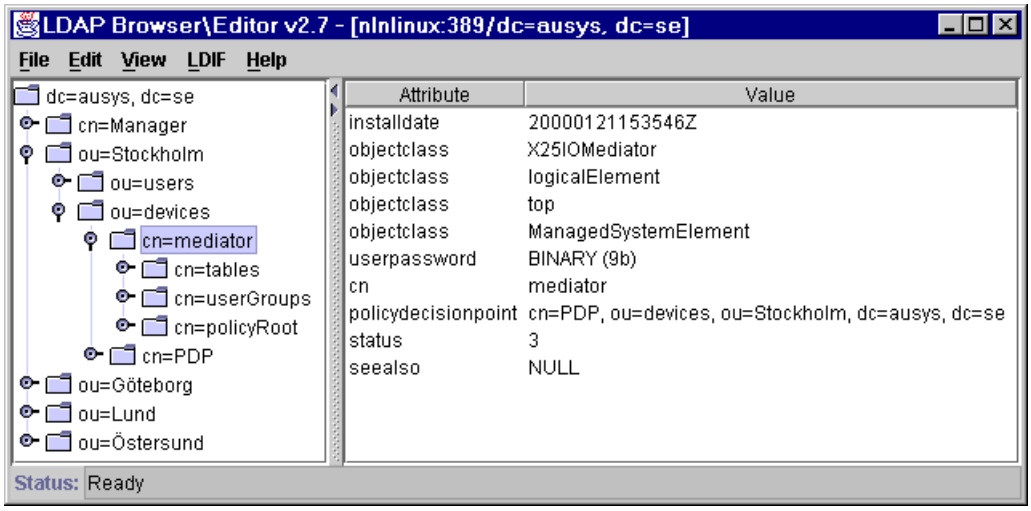

Figure 11. The X.25 I/O Mediator in the DIT..

The right side of Figure 11 shows the different attributes and the values contained in the mediator entry. Also note that the PDP is placed as an entry on the same level and that the DN pointer contained in the 'policyDecisionPoint' attribute points out this entry.

We also define a class for a table that we can place under the X.25 I/O Mediator entry in the directory. It is modelled as a Structural Class, inherited directly from the `top` Class The result is shown below.

```
(<OID> NAME 'table' DESC 'A standard table'
        SUP 'top'
        STRUCTURAL
        MUST( tableName $ nrOfColumns $ columnNames )
        MAY( row ))
```

An example of how a table can look like in the directory is shown in Figure 12 below. Note that we have bound to the mediator entry, and that the hierarchy of entries above the mediator is not shown. The figure shows how the table 'TelnetIn table' with its attributes and values can be represented.

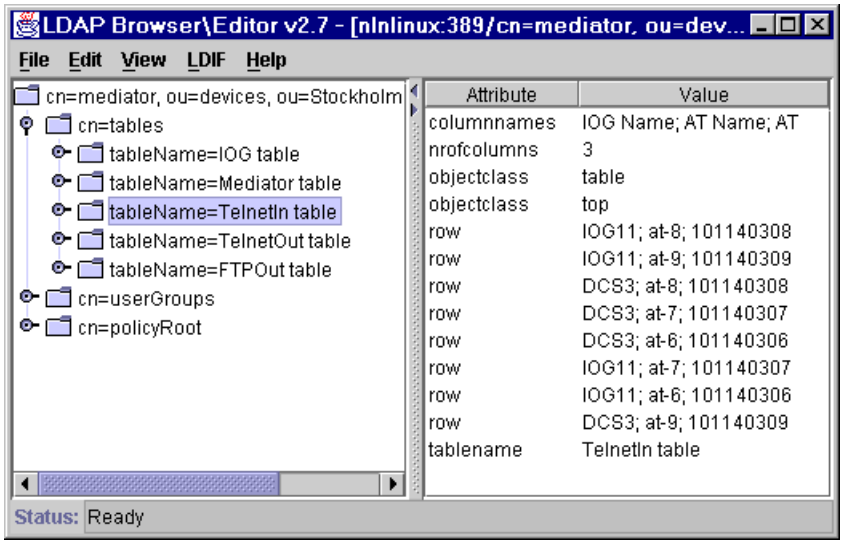

Figure 12. An example of a table in the DIT.

The table consists of four attributes where three of them must exist in the entry. The attributes definition is seen below.

```
(<OID> NAME 'tableName'
        DESC 'A String defining a name'
        SYNTAX DirectoryString)

(<OID> NAME 'nrOfColumns'
        DESC 'The number of Columns in this table'
        SYNTAX Integer)
```

(<OID> NAME 'columnNames'
DESC 'A String defining of the Names in the Column. The names are separated with a ';"
SYNTAX directoryString)

(<OID> NAME 'row'
DESC 'Unordered multivalued attributes describing a row in the table. The values in the attribute are separated with a ';' and must be ordered in the same order as the columNames'
SYNTAX directoryString)

8.4 Defining the Policy Decision Point in the Directory

As for the X.25 I/O Mediator we define the PDP in the directory. The simplified inheritance is used here as well. The important detail for the PDP is to have information in its entry describing how it can be reached. In the solution proposed in this paper the CORBA and IDL standard have been used and therefor the PDP is defined to have an attribute containing information about an IOR file. This IOR file is the file the X.25 I/O Mediator can read and get information about how the PDP can be reached. For further information about the communication between the PDP and the X.25 I/O Mediator, see Section 8.7.1. The definition for the class which describes a PDP is shown below:

(<OID> NAME 'X25IOMediator' DESC 'A Policy Decision Point'
SUP 'LogicalElement'
STRUCTURAL
MUST(IORfile)
MAY())

How it can look like in the directory is shown in Figure 13 below.

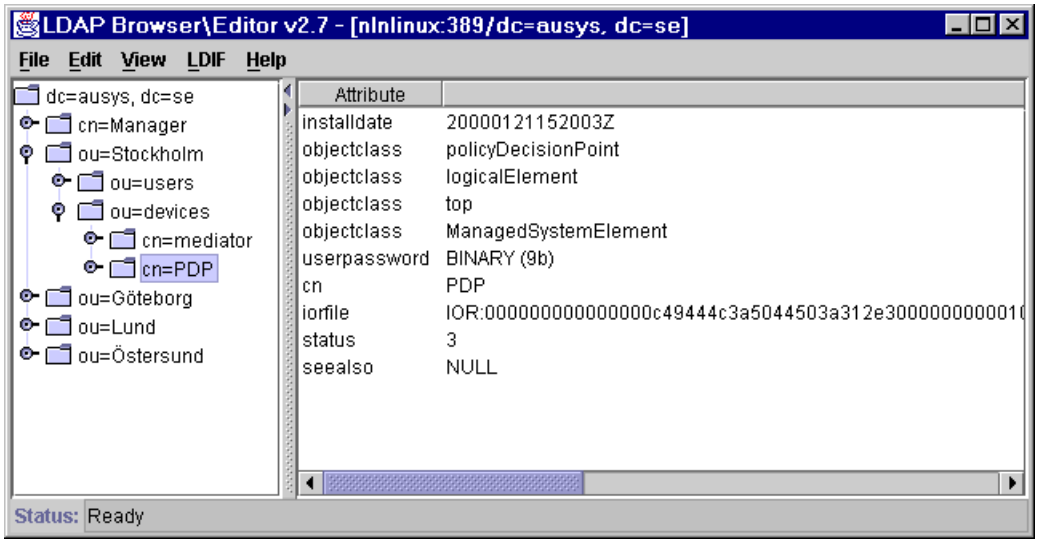

Figure 13. The PDP in the DIT.

Note the cryptic message in the IOR file. This tells the X.25 I/O Mediator all it needs to know, such as IP address, port number etc.

The definition for the attribute IORfile is shown below:

(<OID> NAME 'IORfile'
DESC 'A String defining an IOR file describing how the PDP can be reached.'
SYNTAX directoryString)

8.5 Policy Rules

The information about what users are allowed to do is modelled in the directory with policy rules. A policy rule has attached to it a set of conditions and an action. For the PDP to “execute” the action the conditions in the policy rule must be met. The Core Schema defined by the IETF has defined a set of base classes on how the policy information can be modelled in the directory. The current draft consists of seventeen classes that together can make a PDP retrieve policy information in an efficient manner.

According to the Core Schema conditions and actions are connected to a policy rule via special association classes. Condions are connected via the class policyRuleConditionAssociation and actions are connected via the class policyRuleActionAssociaiton. These associations are DIT contained under the policy rule and the conditions/actions can either be attached directly to the association class since they all are auxiliary classes. Another way is also possible. Instead of adding the conditions/actions directly to the association it is possible to have a DN pointer from the assosiation to an instance of either the class policyCondtionInstance or policyActionInstance. This makes it possible to have reusable conditions and actions. An exception is the class policyTimePeriodConditionAuxClass, which is the only condition given by the IETF in the general model. This class represents the times when a certain policy rule should be active and these conditions are referenced by DN pointers contained in the attribute policyRuleValidyPeriodList of the class policyRule. Using this condition makes it, for example, possible to restrict users to certain IOGs/ATs during certain times.

To fit the X.25 I/O Mediator and the discussion in the previous section the addition of five new conditions to the base schema is proposed. The are the:

- `PolicyTypeOfServiceConditionAuxClass` - which represents what type of service, i.e ftp or telnet/mmlcall a policy rule should apply on.
- `PolicyAuthorizedConditionAuxClass` - which represents what user or group a policy rule should apply on.
- `PolicyIOGConditionAuxClass` - which represents the IOG a policy rule should apply on.

- `PolicyATConditionAuxClass` - which represents the AT a policy rule should apply on.
- `PolicyCommandCondtionAuxClass` - which represent what command a policy rule should apply on.

These five classes derive from the class `policyConditionAuxClass` in the base schema, see Figure 14 below, and not from the class `vendorPolicyConditionAuxClass` which is suggested in the Core Schema.

The motivation to let the classes derive directly from the `policyConditionAuxClass` class is that it simplifies the structure. However, it might have been more correct to let these classes derive from the `vendorPolicyConditionAuxClass`, but on the other hand more complex.

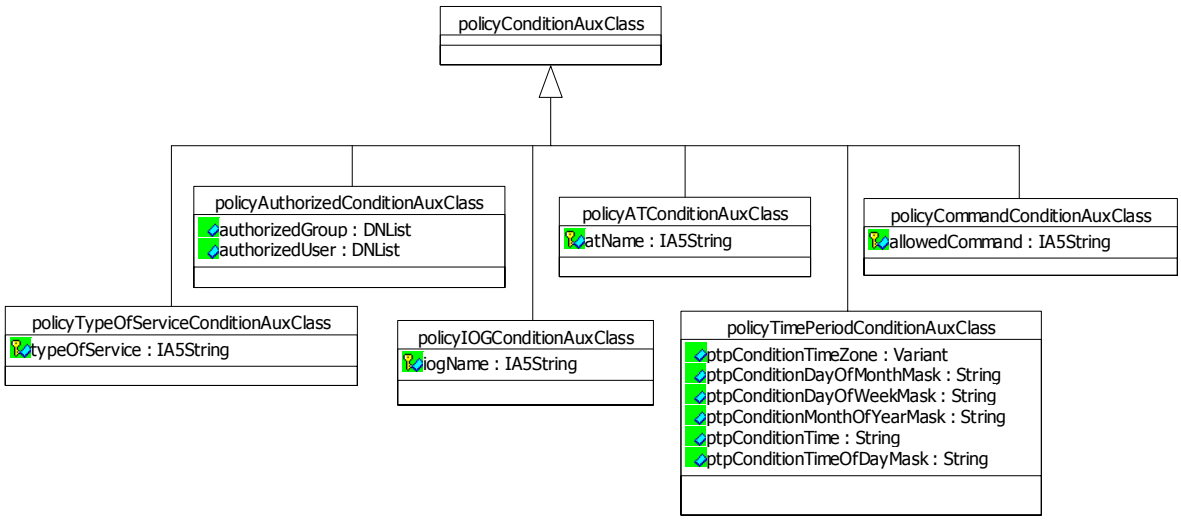

Figure 14. The Policy conditions class hierarchy.

Figure 14 shows the different classes and their attributes. The attributes that are required are shown with a key.

The action that should be taken when the conditions are met is also modelled as a class that is derived from Core Schema. The class is called `policyAuthorizeActionAuxClass`, which is derived from the class `policyActionAuxClass`. It represents a boolean expression TRUE or FALSE which is the answer the PDP returns to the X.25 I/O Mediator.

8.6 How to find the correct policy

The Core Schema does not define exactly how the PDP should receive the policy objects contained in the Directory. It only provides a toolkit of classes to assist the policy Manager as the DIT is being built. There are several different ways the DIT can be built. The basic idea is of course simple. It is to make it possible for the

PDP to retrieve only those policy objects that are necessary, and at the same time use as few LDAP calls as possible. The Core Schema defines that the PDP should be able to do this by searching the directory based on the search filter "(objectclass=policy) ". However it is also specified that the PDP should be able to take advantage of the different toolkits provided by the draft.

This paper investigates two different approaches, which allow the PDP to find the correct policy. They have been implemented with some success but time has not allowed them to be fully examined. A complete LDAP class diagram can be found in Appendix A 'The LDAP class diagram'.

The structure of a single policy rule is, of course, the same in both of the cases, see previous section. The difference consists in the ways of finding the correct policy rule. In both cases the PDP starts to look for policy rules that have conditions that matches the user. Then it searches on groups, if there are any, that the user belongs to.

8.6.1 Subtrees Pointer Approach

The first approach is based on an example in the Core Schema. This solution retrieves a lot of policy information and then browses the objects internally to see if there is any policy rule that satisfies the current conditions.

The PDP is initially configured with a DN pointer to an entry with the "cn=userGroups" contained under the entry "cn=Mediator". See figure 15 below.

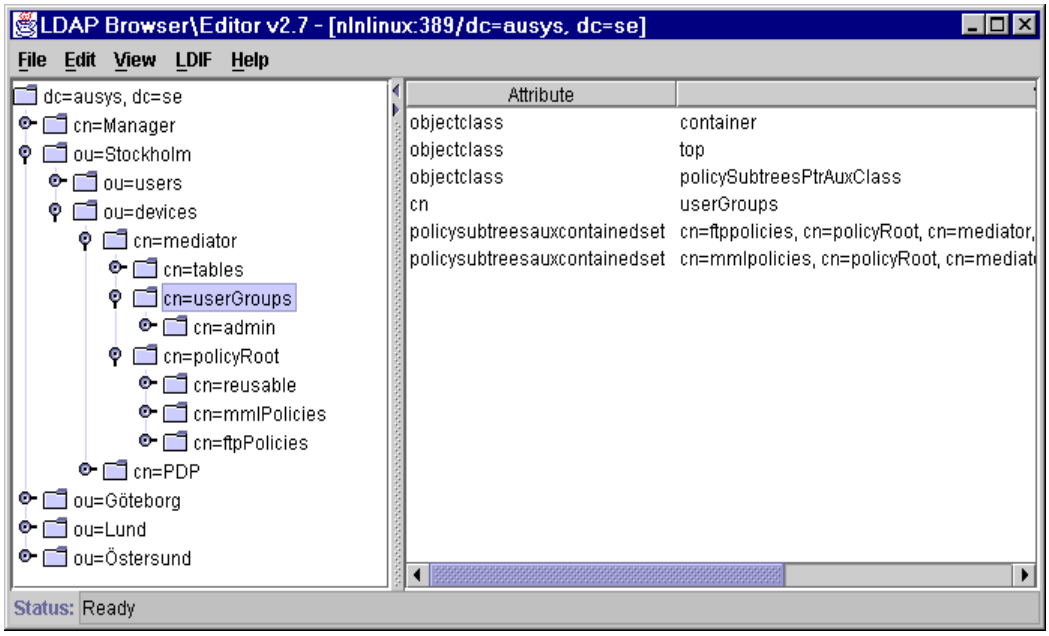

Figure 15. Subtrees Pointer Approach.

The entry "cn=userGroups" is a container that holds information about different user groups configured by the policy Administrator. For example we have the group "cn=Admin", which contains users with administrative rights to the IOGs.

The PDP obtains entries with the searchscope “subtree” to get all the information under the subtree. Hence, the PDP gets information about different groups, i.e profiles, and their members. Another important detail is that the PDP retrieves an attribute of the class policySubtreesPtrAuxClass attached to the entry “cn=userGroups”. This class contains a multivalued attribute called policySubtreesAuxContainedSet. The values in this attribute are DN pointers to subtrees holding information that the PDP needs to make a decision, i.e they point to other containers which hold the policy rules. The PDP then turns to these pointers to collect the policy rules and tries to find a policy rule that fits the arguments sent by the X.25 I/O Mediator. The policy rules are divided into two different groups, each held in a container. The container “cn=mmlPolicies” holds the policy rules which are constructed for Telnet/mmlCall, and the container “cn=ftpPolicies” holds the policy rules constructed for ftp. When the PDP has retrieved the policy rules it tries to match the conditions against the arguments in the request sent by the X.25 I/O Mediator. If the PDP can locate a policy rule that matches the conditions it returns an answer to the Mediator.

The PDP starts to search on policy rules defined for the user. If a rule can not be found the PDP starts to look for rules that have a condition matching a group, if there is one, which the user belongs to. Hence policy rules defined for the user override rules defined for a group that the user belongs to.

This approach is highly effective when one wants to retrieve large numbers of policy rules. The PDP can retrieve almost every rule and then cache them. Thus there is a big chance that the PDP will not need to contact the directory next time it receives a request from the X.25 I/O Mediator. The first time the X.25 I/O Mediator sends a request however, it will probably take a while, depending on the numbers of policy rules, before the PDP finds a rule that fits the arguments in the request.

There is a possibility to obtain fewer policy rules but to still guarantee that the policy rule, which holds the matching conditions, using the Subtrees pointer approach. This involves expanding the PDP’s search criteria, i.e “(objectclass=policy)”. For example, if the PDP recieves a request which looks like `isAllowed(ftp,hja,IOG11,CWD)`, there is a chance that the PDP will still download all the policy rules in the container holding the policy rules with MML/Telnet conditions. Since all that the PDP does is follow the DN pointers to subtrees containing policy rules it might go to the container “cn=mmlpolicies” to download rules first. If the point is not to download a lot of policy rules to keep in its cache this is a waist of time. As mentioned above the solution is to expand the PDP’s search criteria. The abstract class `policy`, which the class `policyRule` inherits from, contains an attribute named ‘policyKeywords’. This attribute can be appended to every instance of the policy rules. Values of this attribute are then used to tag a policy rule with one or several keywords and finally let the PDP include the keywords in the search filter.

This has been done successfully in the implemented prototype. For example we tag every entry of the policy rules with the keywords ‘ftp’ or ‘mml’ depending on the type of policy rule. If the request is the same as the one above, `isAllowed(ftp,hja,IOG11,CWD)`, and we then let the PDP include the first argument in the search, i.e use searchfilter “(&(objectclass=policy)(policyKeywords=*ftp*))”, the retrieved rules will only be the ones that are tagged with the string ftp in the value of the attribute ‘policyKeywords’. Searching like this decreases the number of irrelevant policy rules for that particular request. This is a good idea if the purpose is for the PDP to only retrieve a subset of the available policy rules. Further search criteria can also be implemented. If it is desired, the IOG name can also be put in the ‘policyKeywords’ attribute and thus we have minimised the irrelevant policy rules.

Doing this puts more stress on the directory server when searching the entries, since it involves more search criteria, but on the other hand the returned entries are fewer which facilitates the work the directory server has to do. It is however not recommended that the extended search mechanism be used if the policy rules are expected to not change often, which is normally the case. Then it is better to download a lot of the rules and keep them in the PDP’s cache.

8.6.2 The Containment pointers approach

The second approach is quite different from the first approach. In this approach the manager of the policy rules for the X.25 I/O Mediator must have access to write in the DIT outside the subtree involving the Mediator. This approach is based on the class `policyRuleContainmentAuxClass` and its attribute ‘policyRulesAuxContaintainedSet’, which is an attribute that contains DN pointers to specific policy rules. Moreover, an attribute called ‘memberOfGroup’ is used, which is not an RFC standardised LDAP attribute. It is an attribute used by the University Of Michigan in the class `umichPerson`. The attribute is multivalued and contains DN pointers to groups that the user is a member of.

The PDP is first instructed to search on policy rules attached to the user.

1. Check if the user has any security policies connected to him by checking if any `policyRuleContainmentAuxClass` is connected to the user object. If none is attached, proceed with finding group connected security policies. Else proceed to 2.

2. Find out which security policies are connected to the user by reading the ‘policyRulesAuxContainedSet’ attribute. The values of this attribute contain DN pointers to policy rules.

3. Find out if any of the connected policy rules satisfy the current conditions. If such a rule is found, return the action in the policy rule, i.e TRUE of FALSE, else proceed with finding group connected policy rules.

If the PDP does not find any user connected policy rules, or if the user does not have any rules attached, then PDP searches on the 'memberOfGroup' attribute.

1. Check if the user belongs to any groups, roles, by checking the memberOfGroup attribute. If the user does not have this attribute return by default FALSE. Else proceed to 2.

2. Per group or role, check if any security policies are connected to the group by checking if the `policyRuleContainmentAuxClass` is connected to the group object.

3. Find if any of the connected policy rules satisfies the current conditions. If one is found then return the action in the policy rule, i.e TRUE or FALSE, else return by default FALSE.

Using this approach will make the PDP search the directory many times. For every found pointer to a policy rule it downloads that policy rule. The good thing is that the PDP does not download total irrelevant information. Every policy rule will match the condition in the `policyAuthorizedConditionAuxClass`. It could be questioned if the condition is needed at all since the PDP searches on policy rules containing a matching condition in the `policyAuthorizedConditionAuxClass`. If we have the condition it simplifies the design of the PDP. All the PDP needs to store now are the policy rules with conditions. If we do not have the condition the PDP would either have to access the directory on every request to get what `policyRuleContainmentAuxClass` is connected to what group, or cache those pointers, which might be more complex than having the matching condition attached.

8.6.3 Summary

When comparing the two approaches we see that the first approach downloads a lot of information and then tries to find a policy rule that satisfies the conditions. Proceeding in this manner is a good idea if it is believed that the policy rules does not often change, which normally is the case, since they can be kept in the PDPs cache instead. The second approach on the other hand downloads one policy rule at time and then investigates if the rule satisfies the conditions. Of course the PDP stores the downloaded rules in its cache as well, but in the beginning though it is not believed that the PDP will have the right policy in its cache and therefore it will have to go to the directory to find it.

Note that both solutions searches on the user first. For that reason policies defined for a user will override policies defined for a group or role. It is also perfectly fine for a user to belong to two groups with conflicting policies. It is up to the policy administrator to define what group the PDP should check first. This can be done, in both approaches, by utilising the attribute 'policyRulePriority' in the class `policyRule`. The integer value of 'policyRulePriority' indicates the priority of

the rule. Bigger value of the integer means higher priority. Exactly how this can be done has not been investigated. In the prototype the PDP does not care about the 'policyRulePriority' attribute.

8.7 PDP and X.25 I/O Mediator Considerations

The Policy Decision Point, PDP, as stated in Section 4.2.2, does more than just download information from the directory: It is the PDP that makes decisions based on queries sent by the X.25 I/O Mediator, and that returns decisions made back to the X.25 I/O Mediator. The PDP must, as the directory, provide fast results and always be accessible. Always may be an exaggeration since the X.25 I/O Mediator can be configured to access another PDP if the first one could not be reached.

The PDP in the implemented prototype is written in Java. The reason to make the programming in the Java language is that it is an easy language to understand and handle. There are already classes to download for free that handle LDAP queries. The package used for communication with the directory is the Java Naming and Directory Interface, JNDI, version 1.2.

8.7.1 Communication between X.25 I/O Mediator and the PDP

The communication between the X.25 I/O Mediator and the PDP should, for now, be handled with the CORBA and IDL standard. The current discussion in the DMTF and IETF is that the protocol COPS should be used for this communication. The COPS protocol has recently, during this project, been defined in an RFC. But it is not yet defined if COPS should be the standard way to communicate and therefor the CORBA IDL specification should be used.

The use of the CORBA IDL has the following advantages:

1. It is possible to let the PDP be in a different node, i.e does not have to be on the same machine as the X.25 I/O Mediator.

2. It is a widely supported standard which makes it easy to implement the X.25 I/O Mediator in one language and the PDP in another and still be able to communicate easily without the need to dig down in the transport layer.

3. If a protocol like COPS is to be implemented later, it may easily be done by transforming the Corba PDP service to a local wrapper of the COPS implementation.

The prototype is implemented with help of the free, for none commercial use, ORBacus toolkit. The X.25 I/O Mediator is programmed in C++ and ORBacus supports both IDL to C++ and IDL to Java, which has simplified the communication between the PDP and the X.25 I/O Mediator.

The start up communication for the X.25 I/O Mediator can be seen in Figure 16 below. When the X.25 I/O Mediator starts up it is configured to ask the LDAP directory server, using and LDAP search request, for an IOR file, (1) in the figure. This IOR file contains information about where the PDP is located and how to get a reference to an object called PDPStarter. The directory server returns the IOR

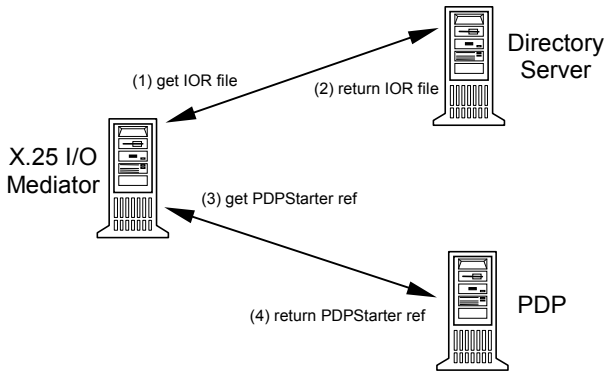

Figure 16. The start up communication for the X.25 I/O Mediator.

file, (2) in the figure. X.25 I/O Mediator can then get a reference to the PDPStarter object, (3) and (4) in the figure, from the PDP.

The continuing communication between the X.25 I/O Mediator and the PDP now continues as in Figure 17. below.

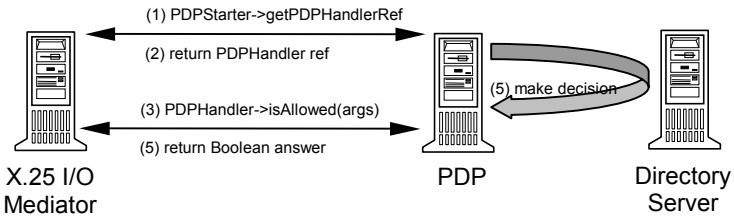

Figure 17. The continuing communication between the X.25 I/O Mediator and the PDP.

For every new client that the X.25 I/O Mediator handles, not shown in the figure, it asks for a reference to a new PDPHandler object from the PDP, (1) and (2) in Figure 17. The X.25 I/O Mediator can then ask the PDPHandler object a question, (3) in Figure 17. The PDP makes a decision based on the arguments sent by the X.25 I/O Mediator, based on cached information or if needed with the help of the directory, (5) in Figure 17, and returns a boolean answer back to the X.25 I/O Mediator. Doing this makes it possible for the PDP to handle several requests simultaneously, since it is possible for the X.25 I/O Mediator to handle several clients at the same time.

8.8 Directory considerations

A crucial device in the solution is, of course, the directory itself. Basically it needs to provide fast results and always be accessible. Therefor the solution proposed

here is that the directory structure is distributed. The directory that the PDP contacts should only contain the necessary information, i.e information about users and the X.25 I/O Mediator. The necessary information could be replicated from a more centralised server.

In the prototype the free directory service OpenLDAP[12] LDAP suite has been used.

8.8.1 The OpenLDAP directory service

In the implemented prototype a directory service called OpenLDAP has been used.

The OpenLDAP project has the effort to develop a robust, commercial-grade, fully featured, and open source LDAP suite of applications and development tools[12]. The current version, latest release, is OpenLDAP 1.2.9. OpenLDAP currently only supports the LDAPv2 protocol and some additional features. LDAPv3 support is going to be included in the OpenLDAP 2.0 release, which seems to be available soon, however not yet.

OpenLDAP builds on the original LDAP project developed by UoM. The directory service consists of a stand-alone LDAP server, called “slapd”, a stand-alone ldap replication server, called “slurpd”, and also some utilites and tools, which can be of use building a directory server.

In this project the directory service has been tested in many different ways.

One important thing the directory service must be able to do, if the solution proposed here is to be used, is to have an opportunity to expand the default directory schema. This has been successfully incorporated. The ojbectclasses and attributes defined by the IETF’s policy framework and the definitions this paper has developed have been added to the existing schema, making it possible to maintain quality in the directory. The schema structure that OpenLDAP uses is possibly a bit too simple if it should be used in a serious attempt to build a directory-enabled network. The definition of what an objectclass must and may contain is seen below. The example used is the class policyRule:

```
# Entry requires objectClass value(s) of:
#                               policyRule
#                               policy
#                               top
objectclass policyRule
                requires
                        objectClass,
                        cn,
                        policyRuleName
                allows
                        policyRuleEnabled,
                        policyRuleConditionListType,
                        policyRuleValidityPeriodList,
```

```
policyRuleUsage,
policyRulePriority,
policyRuleMandatory,
policyRuleSequencedActions,
cn,
cimName,
caption,
description,
policyKeywords
```

As seen above the schema read by the server when starting is really simple. It only tells what the specified class requires and allows. There is no possibility to just add the names the class derives from, to make the server understand what attributes are mandatory. This has to be done manually or with a script. There is also no way to tell the server that a class is abstract or auxiliary. Every defined class is considered by the server to be structural. To be able to maintain a complete directory structure it must also be possible to define where an entry may be placed. For example an entry of the objectclass policyRuleConditionAssociation is only allowed to be placed under an entry of the class policyRule according to the definition in [7]. OpenLDAP currently offers no such possibility.

The attribute definition is also very simple. See below:

```
#definition for the attribute policyRuleName
attribute       policyRuleName                  ces
```

As seen above the name is defined together with the string ‘ces’. This is the syntax definition for the attribute. There are only five syntaxes implemented in the current slap and they are: binary (bin), telePhoneNumber (tel), distinguished name (dn), Case exact string (ces) and case ignore string (cis). This means that there is no possibility to use other required syntaxes. For example an integer must be placed in the directory as either a ‘cis’ or a ‘ces’.

OpenLDAP has also been tried in a distributed manner and that has been working well. References from one directory server have been used to tell a client to contact another and it has worked well. The replication server slurpd has not been tested but it is not believed that it does not work sufficient.

As a summary on OpenLDAP one can say that it is definitely possible to use for simpler tasks right now and when it in the future supports LDAPv3 it can be considered to be used in more advanced tasks. However if one right now should try to use a directory in a larger solution one should choose a commercial product such as Novell´s NDS.

9 Conclusion

The proposed solution in this paper has shown one way of solving some of the security issues concerning the X.25 I/O Mediator, recently released by AU-System. The X.25 I/O Mediator makes it possible for clients placed on a TCP/IP network to communicate with the I/O system on AXE changes placed on an X.25 network. The difficult task of doing this has been to try to base the solution on the forthcoming standardisation Directory Enabled Networks initiative, DEN, currently being developed by the Distributed Management Task Force, DMTF. The goal of DEN is to describe a way of building intelligent networks using a directory as a centralised repository that contains information about everything in the network. The conclusion is though that DEN is not yet ready to be used more than experimentally. There is lots of work left developing the DEN standard. Many companies are supporting the standardisation and while agreeing that DEN is needed there are big differences in how the standard is supposed to be defined. It seems as if the standardisation process actually has slowed done completely.

The first goal of the project was to base the entire work on DEN, but this seemed impossible. Therefor, the solution is based on an internet draft made by the Policy Framework Working Group in the Internet Engineering Task Force, IETF. The group has tried to develop a general policy information model that describes how policies can be stored in a directory. However, the work going on in this group is also far from ready and it is not believed that the structure of the classes is the final one. There are still many different opinions on how policy information should look like. The draft does not define one explicit way of how the information should be organised in the directory. It defines a set of classes that can be used instead. This paper has presented two different solutions of directory structure. One of them has been implemented and tested with some success. It is not believed that these are the only way to organise the structure of policy information, nor is it believed that it is the best way.

It is not assumed, either, that it is a good idea presently to try to use a DEN concept in the network. Therefore, it must also be said that the use of the solution proposed in this paper is not efficient enough if one does not use application specific methods. The idea is not completely useless, though. LDAP based directories are already being used, for example in Microsoft Windows 2000's Active Directory and Novell's NDS. Once the DEN standardisation is completely developed, policy based networking will probably be a great help in network management and therefore broadly applied.

10 Abbreviations

API	Application Program Interface
COPS	Common Open Policy Service
DAP	Directory Access Protocol
DAS	Directory Assistance Service
DEN	Directory Enabled Networks
DIT	Directory Information Tree
DIXIE	Directory Interface to X.500 Implemented Efficiently
DMTF	Distributed Management Task Force
DN	Distinguished Name
FTP	File Transfer Protocol
I/O	Input/Output
IETF	Internet Engineering Task Force
IOG	Input Output Group
LDIF	LDAP Data Interchange Format
LDAP	Lightweight Direct Access Protocol
MML	Man Machine Language
MTP	Messasge Transfer Protocol
OS	Operating System
OSI	Open Systems Interconnection
SDK	Software Development Kit
SNMP	Simple Network Management Protocol
SP	Support Processor
SSL	Secure Sockets Layer
TCP	Transport Control Protocol

11 References

[1] Howes, Timothy A., Good Gordon S. and Smith, Mark C. 1999. *Understanding and Deploying LDAP Directory Services*. ISBN 1-57870-070-1, Macmillan Technical Publishing.

[2] Wahl, M., *A summary of the X.500(96) User Schema for use with LDAPv3*, RFC 2256, December 1997.

[3] Howes, T., Kille S., Wahl M., *Lightweight Directory Access Protocol (v3)*, RFC 2251, December 1997

[4] Howes, T., *LDAP: Use as Directed*, February 1999, http://www.data.com/issue/990207/ldap.html

[5] Semeria, C., Fuller, F., *Directory-Enabled Networks and 3Com's Framework for Policy-Powered Networking*, white paper from 3Com, http://www.3com.com/technology/tech_net/white_papers/500665.html

[6] Policy Framework Working Group, *General Homepage*, http://www.ietf.org/html.charters/policy-charter.html

[7] Policy Framework Working Group, *Policy Framework LDAP Core Schema*, http://www.ietf.org/internet-drafts/draft-ietf-policy-core-schema-06.txt, November 1999

[8] LDAP Extension Working Group, *General Homepage,* http://www.ietf.org/html.charters/ldapext-charter.html

[9] Ericsson Telecom AB, *Operation of SP-based IO Systems*, 1994 Stockholm Sweden.

[10] Suns Homepage concerning the Java Programming Language, *General Homepage*, http://java.sun.com

[11] Netscape Directory SDK, *General Homepage*, http://www.mozilla.org/directory

[12] The OpenLDAP Project, *General Homepage*, http://www.openldap.org

[13] Suite Software, *Security Framework, White Paper*, 1997, http://205.153.188.171/suite.com/whitepapers/wp6.html

[14] Judd Steve, Strassner John, *Directory Enabled Networks–Information Model and Base Schema version 3.0c5*, 1998-09-28, http://www.murchiso.com/den/specifications/directory-enabled-networks-v3-lastcall.pdf

Appendix A: The LDAP class diagram