From SNMP deception to VeriNeC's Cisco service

Master Thesis in TNS Reseach Group

Departement of Informatics
University of Fribourg, Switzerland

Referent: Prof. Dr. Ulrich Ultes-Nitsche
Assistants: Dominik Jungo, David Buchmann

Author:
Christoph Ehret
Bd de Pérolles 81
1700 Fribourg
christoph.ehret@unifr.ch

30th September 2005

Abstract

This paper reports on my Master Thesis, which presents in a first part my observations on the way how the Simple Network Management Protocol is integrated in network devices, and in a second part my implementation of a Cisco translation service in the project called Verified Network Configuration (VeriNeC) [13] funded by the Swiss National Science Foundation. In today's growing number of network devices, we really need an easy and efficient way to manage all these different devices. SNMP is theoretically a good candidate since it is the standard management protocol, but practically it is unfortunately misused by the network devices vendors. With VeriNeC, we can even go further than just management. The entire network definition and configuration is represented in an eXtensible Markup Language (XML) document which can be tested by its simulation module. After successful tests, the XML abstract definition of the network is translated into a configuration specific to a device and transfered to it. The translation is done using eXtensible Stylesheet Language Transformations (XSLT). In my work, we can see how to implement a translation service for Cisco.

Keywords: VeriNeC, SNMP, Cisco, Ethernet, Packet-filters, Routing

Contents

List of Figures

Chapter 1

Introduction

Did you say *deception*? What has the word *deception* to do with the simple network management protocol? Would *disappointment* not be a more appropriate word? If these are the questions you had when you read my Master thesis title for the first time, I must admit that you are completely right. At the beginning, this was actually my fault, since I gallicized the word *deception* which has not the same meaning in English as in French. Once a friend of mine made me conscious of that language error, I was considering again my title thinking that *disappointment* would certainly be a better choice. But after thinking a while, I finally maintained the original title, deciding that *deception* implies somehow *disappointment* and if we look at what we will *experience* with SNMP during the whole documentation, I am sure that you will approve my choice.
If we come back to the meaning of the title, can you feel the impression of movement? Can you see the idea of a starting and ending point? I am sure, you certainly understand what I want to express with the title "From SNMP deception to Cisco's VeriNeC service", don't you? With this title I want to show that my initial project goal was to study the possibility to implement a SNMP distribution service for VeriNeC, but that I found myself stuck in a dead end because of the unfeasibility of the project goal. From that moment my project completely changed and I finally ended up with the implementation of VeriNeC's Cisco (translation) service. The title also contains all the different parts of the project, which we will discuss throughout this documentation, namely *SNMP*, *Cisco* and *VeriNeC*. VeriNeC is a rather complex framework, Cisco uses a very complex system; how will it be possible to combine them together?

The first part of the documentation will focus on the simple network management protocol, well known under its abbreviation *SNMP*. We will begin to explain what SNMP is and see the different "actors" and components of this management protocol. Since we speak about management, we will then have to speak about the managed data, how we can represent it, how and where we store it, how we can access this data through a network. Once we are through these more technical parts of the protocol, we will discuss how good (or bad) SNMP has been integrated in vendors network devices; this is indeed a sign that will show us how much the vendors trust in this management protocol. We will end this chapter on a discussion about the future of the simple network management protocol. This chapter will explain you why my initial project, called *SNMP Research*, was finally not feasible and which important role SNMP can play in todays growing number of interconnected network devices.

Is SNMP really "the" management protocol? Is it really as promising as it is in theory? How good is SNMP supported in network devices vendors' systems? How can we be disappointed by SNMP? These are all questions we will answer in the first part.

Chapter 3 will give us an introduction to VeriNeC; this chapter introduces everything we will need for a good comprehension of my Master thesis. We will first explain what VeriNeC is and see the different modules that compose it. We will then see the concept of abstract configuration document, which we can call the central point of VeriNeC. Finally, we will focus on the translation module, since this is the part of VeriNeC that I use in my project and have to extend with my Cisco service; this last part will explain the notions of *translator*, *restrictor* and *XSL Repository* what will often be used in the next chapter. What makes VeriNeC so interesting to use in heterogeneous networks? What can VeriNeC do to improve the security of networks? Some questions that will be interesting to discuss in this third chapter.

The fourth chapter will present my Master thesis and all the different concepts I had to work with. We will begin this chapter with the enumeration of the different requirements I had to achieve, followed by the explanation of the different technology elements I used to carry the requirements off. We will then come to the most important part of this chapter, namely the implementation. We will first begin to explain everything that has to do with Cisco, i.e. how it is possible to distribute a configuration file to a Cisco device and what the syntax and the content of such a configuration file do look like. After that, we will see the different translators that will be used in the translation process for a given service, and the schema that is extended to take my specific distribution parameters into account. The last part of this important section is dedicated to the Java implementation, where we will discuss all the classes that are used for my distributor. At the end of this chapter, we will understand how my distribution service for Cisco is implemented, what is needed to make it work and what are generally the different components needed to implement our own distribution service. We will also discover throughout the whole chapter the problems and difficulties I encountered and the solution I took to solve them. We will close Chapter 4 with possible improvements and critics on my Master project.
How easy can we distribute a configuration to a Cisco device? What are the best strategies and technologies to choose to have a Cisco translation service as efficient as possible? How many Java classes will we need to implement the Cisco distributor? How good will my VeriNeC Cisco translation service finally be? These are all questions we will discuss in the fourth chapter.

Chapter 2

SNMP

In this chapter, we will see what the **S**imple **N**etwork **M**anagement **P**rotocol is. This chapter will also explain a part of the title of my Master Thesis : at the beginning of my work, the title was indeed *SNMP Research*. I had to see if it was possible to implement a SNMP distribution service to *VeriNeC*[13]; unfortunately, the results of my research were not as we expected and another way had to be found. We will understand from section 2.6 why the title of my Master thesis changed from *SNMP Research* to *SNMP deception*[1]. Before we come to this section, we should first take a closer look at SNMP, when it was created and what its different components are. As we speak about management, the information plays a capital role; we will see how this information is structured, i.e represented, and what it actually contains. Like with every database, we need some operations to access, retrieve and modify the data; we will see that SNMP also provides the user with a set of operations to read and modify the managed information. Once we are through this technical part, we can understand the *disappointment* I had and the conclusions I had to take from my research about SNMP and its implementation by the different vendors. To finish, we will shortly speak about what could be the future of the **S**imple **N**etwork **M**anagement **P**rotocol.

2.1 What is SNMP ?

In today's World of complex networks, where routers, printers, switches or servers are standard components, we really need a simple, but efficient way to manage them, to be sure all the devices are running correctly and performing optimally. The **S**imple **N**etwork **M**anagement **P**rotocol (SNMP) is exactly what we are looking for : as we can read, it is a simple protocol for managing and monitoring different Internet Protocol devices on a network. The protocol provides the users with a simple set of operations (see section 2.5). As we will see, there are different versions of SNMP.

The Simple Network Management Protocol was introduced because a standard to manage the growing number of network devices was needed; so appeared in 1988 the first RFCs for SNMP, now known as **Simple Network Management Protocol version 1**, shortly **SNMPv1** :

[1]This is of course only the first part of the title

- **RFC 1065** - Structure and identification of management information for TCP/IP-based internets (now obsoleted by RFC 1155)
- **RFC 1066** - Management information base for network management of TCP/IP-based internets (now obsoleted by RFC 1156)
- **RFC 1067** - A simple network management protocol (now obsoleted by RFC 1157)

This version was widely criticized, because of its poor security. The authentication of users or hosts is based on a *community string* that we can compare to a kind of password; this *community string* is sent in plain-text, like in FTP, what implies that it can be very easily caught. In SNMPv1, there are typically three different types of communities:

- read-only
- read-write
- traps[2]

With this security problem in mind, the community tried to move toward a new SNMP version.
SNMPv2 revises Version 1 and includes improvements in the areas of performance, security and confidentiality. However, the new security system for Version 2 was actually so complex, that at the end it was not widely accepted. Because of this complex security framework, the actual Version 2 standard is known as *Community-Based Simple Network Management Protocol version 2*, or simply *SNMPv2c*; it is SNMPv2 without the controversial security model, using instead the simple community-based security scheme of SNMPv1. SNMPv2c is presently widely supported by the vendors and different SNMP implementations. SNMPv2 and SNMPv2c are defined in RFCs 1441, 1448, 1449, 1901, 1905, 1906, 1907 and 1908.
SNMP Version 3, defined in more than ten RFCs, finally resolves the security weakness we have in the first and second version and has been recognized since 2004 by the *Internet Engineering Task Force* (IETF) as the current standard version of SNMP. This standard version only resolves the security issue, but makes no other changes to the protocol; that is the reason why in practice all three versions of the protocol can easily coexist, and this point of coexistence is defined in RFC 3584. Version 3 adds support for strong authentication, using MD5[3] or SHA[4] algorithms to authenticate users, and privacy, using DES or AES algorithms to encrypt and decrypt SNMP messages.
We have seen so far what SNMP is and its different versions. We will now have a closer look at how the protocol works and what are its components.

[2]The trap community string allows to authenticate the traps (asynchronous notifications) sent by the agent.

[3]Since 2004, MD5 is not secure anymore or inappropriate for certain uses, due to a known collision weakness: `http://eprint.iacr.org/2004/199.pdf`

[4]This algorithm has also been broken at the beginning of 2005 by the same research team that had broken MD5 : `http://cryptome.org/wang_sha1_v2.zip`. This algorithm is today still more secure than MD5.

2.2 SNMP components

In SNMP world, there are two different *actors* : *managers* and *agents*. The manager, often referred to as *Network Management Station* (NMS), is responsible for sending queries to the agent in order to receive some piece of information and receiving traps from agents in the network. Traps are only sent by the agents and when a specific event on their network device happens, for example when there is a hardware problem or when the amount of traffic exceeds a certain level. When the NMS receives a trap, it will take some kind of action, like (for example) sending an SMS to the system administrator's mobile phone to inform him that there is a hardware problem on a given network device, or take some specific measure to slow down the traffic.
At the other end, we have the agents. An agent is a program that runs on each managed network device; it is responsible for sending responses to the NMS's requests and traps when a special event happened. The agent always knows the exact state of its device and which information it can send or not; the agent is a kind of *gatekeeper*, as we can see it on Figure 2.1, because it is the only one who has a direct access to the Information Base, i.e the global state of the device (see Chapter 2.4). The agent can be a separate program, running as a daemon under Unix for example, or it can be incorporated into the operating system, like Cisco's IOS.
As we can see it on Figure 2.2, we can compare the way the manager and the agent communicate to the client-server model : the manager, which represents the client, is a program that sends requests to the agent, while the latter, representing the server, processes the request and sends an answer back to the manager[5]. What is special here, is that we have more *servers* than *clients*.

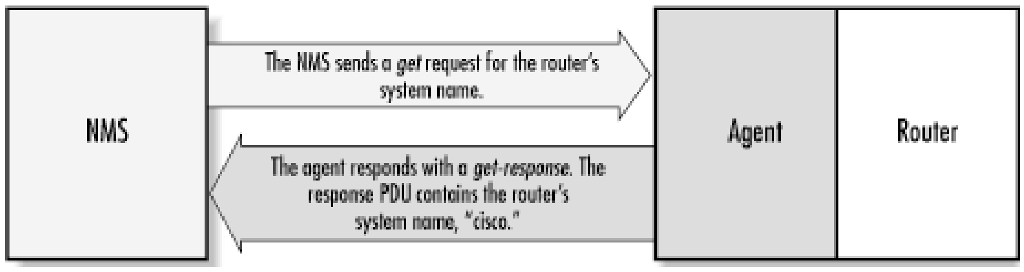

NMS
The NMS sends a *get* request for the router's system name.
The agent responds with a *get-response*. The response PDU contains the router's system name, "cisco."
Agent
Router

Figure 2.1: The agent as *gatekeeper* [1]

The transport protocol that SNMP uses does also have its importance in the design of our management protocol. It uses UDP as transport protocol on port 161 for sending and receiving requests, and port 162 for receiving traps from managed devices; we can see this on Figure 2.3. Why UDP and not TCP? First we have to remember that when administration is needed, then there could certainly be a problem, due to a heavily congested network for example. UDP is connectionless, what means no end-to-end connection is made between the agent and the NMS when they communicate together, in the contrary to TCP. UDP requires low overhead, so the impact on the network's performance is reduced, which is not to be neglected when we have to manage an overloaded network. Imagine we would use TCP in a heavily congested network : it would lead to a situation where the network would be flooded with retransmissions in TCP's attempt to achieve re-

[5]This representation is of course not true for traps.

liability, what would have a bad impact on an already congested network. The unreliable aspect of UDP, as it does not send acknowledgments for every packet, means that it is up to SNMP to determine if packets are lost and if so to send them again; to achieve this, it uses a timeout. The NMS sends a UDP request to an agent and wait for a response; if the timeout is reached, the NMS assumes the packet was lost and retransmits it. The time interval the NMS waits for a response before a timeout occurs and the number of times it retransmits a packet is configurable. The unreliability of UDP is a bit more problematic for traps : if an agent sends a trap, it has no way to know if it has arrived at the NMS or was lost, as the NMS does not send an acknowledgment back to the agent and the agent itself does not wait for a response from the NMS; and of course it is impossible for the NMS to know that the agent has tried to send it a trap if the packet gets lost.
Now, we know how the Simple Network Management Protocol works and what are its components. It is time to have a look at *management information*, i.e. what it contains and how it is represented.

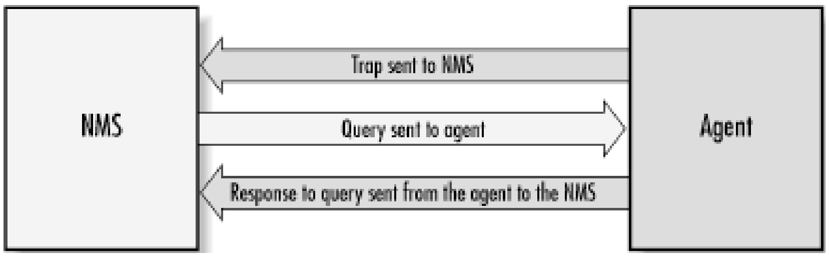


Figure 2.2: Manager-Agent as Client-Server model [1]

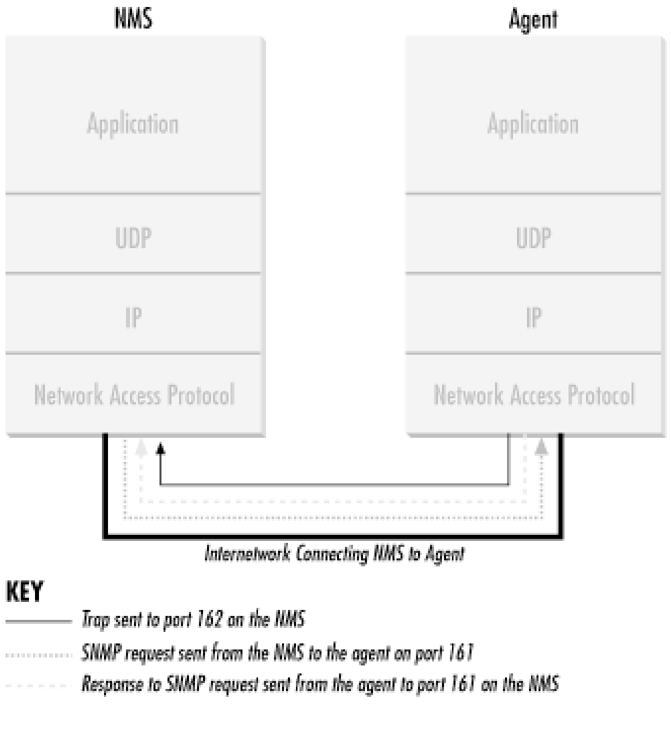

NMS
Agent
Application
UDP
IP
Network Access Protocol
Application
UDP
IP
Network Access Protocol
Internetwork Connecting NMS to Agent
KEY
Trap sent to port 162 on the NMS
SNMP request sent from the NMS to the agent on port 161
Response to SNMP request sent from the agent to port 161 on the NMS

Figure 2.3: SNMP in the TCP/IP protocol suite [1]

2.3 The Structure of Management Information

To be able to understand what kind of data a device can provide, we first have to see how this data is represented within SNMP. The *Structure of Management Information* (SMI) defines how *managed objects*[6] are named and specifies their associated data type. There are two different versions of SMI :

- SMIv1: The original version of SMI defined in RFC 1155 and used by SNMPv1
- SMIv2: Provides enhancements for SNMP Version 2 and 3, and is defined in RFC 2578

The definition of managed objects can be divided into three attributes:

Name The name, or the *Object Identifier* (OID), uniquely defines a managed object. The name can be represented either in a numeric form or in a more *human readable* way. As we will see later, both are not very convenient.

Type and Syntax The data type of a managed object is defined using a subset of *Abstract Syntax Notation one*; ASN.1 is a notation that specifies how data is represented and transmitted between managers and agents. This abstract notation is also platform independent, which is the reason why it can be used without any problem on any device possible. We will see more on this topic later in this section.

Encoding The *Basic Encoding Rules* (BER) are used to encode a single instance of an object into a string of octets. BER is quite suitable for sending the object on a transport medium.

We will now have a deeper look inside the naming of OIDs and into their syntax.

Naming the managed objects

To understand the naming of the managed objects, we first have to see that they are organized in a tree-like structure, like we can see it in Figure 2.4. An object identifier is made up of a serie of integers separated by dots, where each integer represent a node in the tree-like structure. If we have a look at Figure 2.4, we see that the node *mgmt* (for management) is represented by `1.3.6.1.2` which is the path to this node. As we already told it, we can also describe the path to a node in a more *human readable* way; if we again take as example the node *mgmt*, instead of using the serie of integers, we could use the names of each node represented by an integer : `iso.org.dod.internet.mgmt`. As we can see, it is not very convenient to describe a long path with one or the other form. The SMI definition for the *mgmt* node would be the following :

```
internet    OBJECT IDENTIFIER ::= { iso org(3) dod(6) 1 }
mgmt        OBJECT IDENTIFIER ::= { internet 2 }
```

[6]Under managed objects, we mean all the objects in a device that can be managed, like the MAC address of an ethernet card in a router or the entries in the routing table

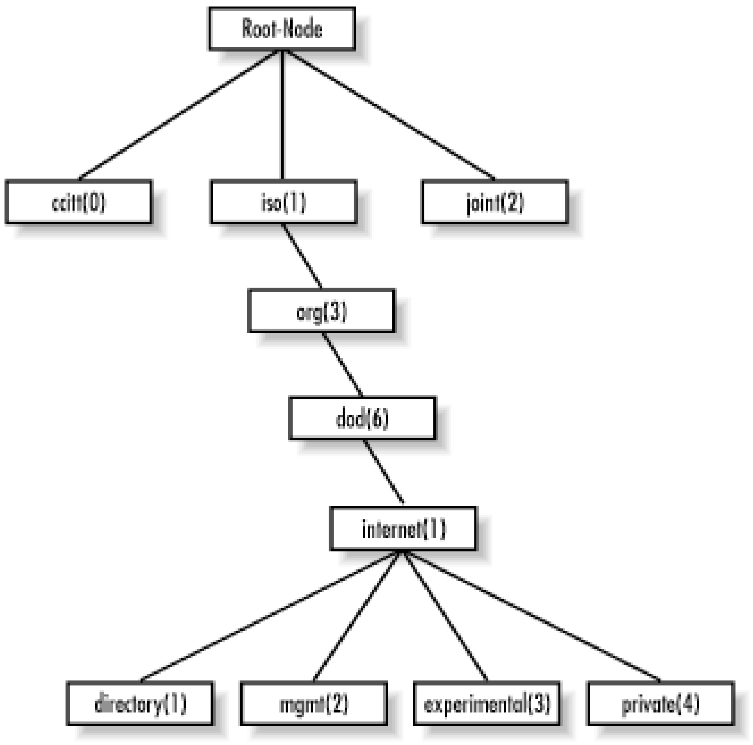


Figure 2.4: Tree-like hierarchy of SMI

Before ending this naming part, I just want to mention an important branch under the *private* subtree. It is used to give hardware and software vendors the possibility to define their own private objects for their hardware or software they want managed by SNMP. We do not see this subtree on Figure 2.4, but SMI defines it as :

```
enterprises    OBJECT IDENTIFIER ::= { private 1 }
```

The path to this node would be given by `1.3.6.1.4.1` or with the more human readable way, *just* `iso.org.dod.internet.private.enterprises`. Every enterprise, institution or organization that is interested to have a private branch to put all its private objects in it, i.e private MIBs (see section 2.4), can register freely at the *Internet Assigned Numbers Authority* (IANA) to be assigned a private number. *Cisco* for example has been assigned number 9, so all its private ojects space can be found under the path `1.3.6.1.4.1.9` or simply `iso.org.dod.internet.private.enterprises.cisco`. A list of the current assigned numbers can be found at `ftp://ftp.isi.edu/in-notes/iana/assignments/enterprise-numbers`.

Syntax

The naming of the managed object with SMI is one part, and the other part is the syntax, the way how to represent and define the objects. As we already said it, a subset of ASN.1 is used to specify how data is represented and transmitted between agents and managers. Like the computer programming language Java or C, SMI defines several data types that tell us what kind of information a managed object can hold. SMI Version 2 extends

the data types that were defined with SMIv1. Here are a few data types from SMIv1 : `INTEGER` , `OCTET STRING`, `IPADDRESS`, `OBJECT IDENTIFIER`, `SEQUENCE`, etc.
In the next section about MIB, we will see an example that shows the use of this syntax.

2.4 Management Information Base

We have just seen in the last section that SMI provides a way to define managed objects. Now we need a way to store or gather all these objects in a single place. This is exactly what the *Management Information Base*, shortly *MIB*, does.
We can compare the MIB to a database of managed objects that the agent tracks. Thanks to the information base, the NMS knows the objects it can access and the actual status of them. MIB is the definition, using SMI syntax, of the objects and also gives the meaning of each object; we can compare it to a dictionary, where we can find the definition and meaning of a word. Most of the MIBs, above all the vendors one, are distributed as human-readable text files, so that they can be read and inspected with a normal text reader program. Thanks to this, every vendor can distribute the different MIBs that specify all its objects that can be managed with SNMP; Cisco for example has hundreds of different MIBs for all its product lines.
Every object in a MIB file is defined as follow :

```
<name> OBJECT-TYPE
    SYNTAX <datatype>
    ACCESS <read-only | read-write | write-only | not-accessible>
    STATUS <mandatory | optional | obsolete>
    DESCRIPTION
        "Textual description describing this particular managed object."
    ::= { <Unique OID that defines this object> }
```

The *ACCESS* field is rather important for the managers, that is they have to know the access permission to a managed object; some are *read-only* and maintain for example some status information, some are *read-write* what means we can obtain some values from them and modify them, others are *write-only* like objects where we specify for example which protocol we want to use in a file transfer, and finally we can have *not-accessible* for objects that are only used inside the MIB by the agent.
The following example of an object from a human-readable MIB file[7] will show us how it looks like and give us also a view of SMI syntax :

```
ccmHistoryEventCommandSourceAddress OBJECT-TYPE
        SYNTAX IpAddress
        ACCESS read-only
        STATUS current
        DESCRIPTION
            "If ccmHistoryEventTerminalType is 'virtual', the internet
            address of the connected system.
```

[7]Extract of the CISCO-CONFIG-MAN MIB

```
        If ccmHistoryEventCommandSource is 'snmp', the internet
        address of the requester.

        The value is 0.0.0.0 if not available or not applicable."
    ::= { ccmHistoryEventEntry 10 }
```

A human-readable MIB file is full of object declarations, but also with a lot of other parameters we will not discuss here. The managed device has of course a compiled version of these MIB files installed in it and all these MIB files make up its Management Information Base.

2.5 SNMP Operations

SNMP provides the user with a set of operations that permits the managers and agents to communicate and exchange messages. We will not explain all the operations in details, but only give a little overview of each, and see at the end of the section an example of one operation using one implementation of SNMP.

As we can see it in Table 2.5 on Page 17, all the operations are quite simple. The table also shows us that some operations only appeared with SNMPv2.
We will finish this section, by showing a little example with a *set* operation; we will use the *Net-SNMP*[8] implementation in this example :

```
snmpset -v 2c -c myprivateCommunityName 10.10.10.1
        1.3.6.1.4.1.9.9.96.1.1.1.1.2.333 i 1
        1.3.6.1.4.1.9.9.96.1.1.1.1.3.333 i 1
        1.3.6.1.4.1.9.9.96.1.1.1.1.4.333 i 3
        1.3.6.1.4.1.9.9.96.1.1.1.1.5.333 a 10.10.10.2
        1.3.6.1.4.1.9.9.96.1.1.1.1.6.333 s running-config
        1.3.6.1.4.1.9.9.96.1.1.1.1.14.333 i 4
```

From the above example – this is actually the way it works with every SNMP implementation – , we can see that we have to specify : first the SNMP version we want to use, followed by the community name (for Version 1 and 2) and the IP address of the agent we want to query; for Version 3, instead of the community name, we would have some authentication and privacy parameters. Finally we can see all these series of integers, that are the OID in the MIB we want to modify or create. As we can see, the command is quite long and we can only hope that we made no typing mistake.

2.6 SNMP and the vendors

We have seen so far, that the *Simple Network Management Protocol* is the standard protocol to manage IP based network devices; it provides the following features :

[8]Net-SNMP is a standard package on Mac OS X and most Linux distributions :
`http://net-snmp.sourceforge.net`

- Get the status and different information from network devices and applications
- Be informed by agents when special events on a network or application happened
- Configure managed devices or applications
- Strong authentication and privacy with Version 3

SNMP should actually be a dream, the *Holy Land* of every network administrator; even if they have a lot of different network devices from different vendors to manage, they just have to read the corresponding MIBs and see which objects they need to manage. The administrators do not need to know the architectures and the way all these different devices works, as they just have to read the MIBs that are all written in standard SMI syntax. This situation remembers a well know citation from a well know book and film : *One to rule them All*[9] .
We can easily imagine how interesting this would be for network administrators... But the reality is unfortunately not like that. SNMP is widely supported by network devices vendors, but they only use traps and the monitoring part of the Simple Network Management Protocol; the configuration part is very poorly supported, at least it is never possible to configure every parameter of a device using SNMP, like on Cisco devices for example. Even if authentication and privacy are very important in communication today, we still have some devices and applications that do not support SNMPv3.
Despite all we have seen, we can ask ourselves why we do not have a full and total support for all the features provided by SNMP? There are two main reasons for this :

1. Business
2. Proprietary solutions are provided to users

Imagine a vendor has a quite complex and huge system to configure its devices; imagine this same vendor would fully support and integrate SNMP to configure its devices. Now, if a network administrator has to choose between something (s)he is already familiar with as (s)he just has to read the MIBs and find the needed objects for configuration, and something completely new to her/him, what will (s)he choose, if the result will be the same at the end? Even if the system is really good and comes with a lot of features, the administrator would choose the solution using SNMP. The vendor would have developed the whole system for nothing, and there would be no possibility to make money with it. For a vendor, it is also simpler to just develop a proprietary system than to follow standards and be dependent on standards consortium. So, the best solution for the vendor is to suppress the configuration property of SNMP and to give the users no other choice than learning how the system works. As the system is really complex, the vendor will propose some special (expensive) courses or certifications, where the users will learn everything about the system. Once an administrator has invested a lot of time and money to learn how the system works, he feels confident with this system, will buy it and will continue to use it in the future.
For the courses or certifications you also need some documentation, so the vendor can again make some money with writing books. At the end, the vendor has really created a

[9]From the book *The Lord of the Rings* written by J.R.R. TOLKIEN

whole business around his complex system. We can write the following and very interesting equation to show this :

$$\text{No configuration via SNMP} = \text{Money}$$

This is indeed exactly what *Cisco* (see Appendix A) does : they have written a complete Operating System, known as *Cisco IOS*, to manage their devices, and for the same version of the IOS, each device or series of devices can have some minor differences in its IOS. Cisco IOS is quite complex and provides a lot of features, and every *type*[10] of devices has some special commands. If you really want to understand how the IOS works and how to use it, the best thing you can do is to get one or several Cisco certifications, depending on the *type* of devices you will need on your network; today, you really need to be Cisco certified if you want to find a job as network administrator in a big enterprise network. To have an idea on how good this certification business is[11], we can have a look at some examples[12] :

- The exam prices range from $65 to $300 US dollars
- Lab exams cost $1250 US dollars

As we can see, it is quite a lucrative business. To this, we can add the books that are recommended for the exam's preparation, prices[13] range from $30 to $120 US dollars. Of course, this is only one supposition why some vendors like Cisco do not fully implement SNMP in their system.

This leads us to another (possible) reason, why vendors prefer to not implement the configuration ability of SNMP : they prefer the users to use their own solutions to manage their network devices. Cisco for example provides free tools that are really great and widely used to manage their devices[14]. A common solution used presently by almost every network devices vendor is to provide a web-interface to configure quickly and easily the most important parameters of the device. For *every day* users who do not need to configure further the device, this solution is sufficient, but what about the network managers? Again, they have to use the command line interface (*CLI*) if they want to change specific parameters of the device. Actually, the web-interface – often applets that are loaded to the user from the device – simply emulates the CLI commands. Of course, we can read here or there, or see some models, where the web-interface emulates SNMP commands to configure the device; this means that the configuration part of SNMP is supported, but it is impossible to find more documentation about this. By the way, I have not found any device that has this kind of configuration.
It seems that every vendor likes to reinvent the wheel again and again, preferring to do his own cooking and business regarding the way to manage their devices or operating system.

[10]With *type*, we mean firewalls, routers, VoIP solutions, IDS, IPS, etc.

[11]Cisco certifications have a very good reputation, are worldwide recognized, and are technically rather difficult

[12]All the examples can be found on `http://www.cisco.com`

[13]Prices found on `http://www.amazon.com`

[14]One example is *Cisco Network Assistant*, a network management application, unfortunately only working on Windows : `http://www.cisco.com/en/US/products/ps5931/index.html`

2.7 SNMP and the future

After what we have seen in last section, it is clear for everyone that the future of SNMP will mostly depend on the vendors. We can read on a lot of vendors support section and forums on the Internet, that a better support of SNMP is promised in the future. What have we to understand with *a better support of SNMP*? Does this mean more monitoring possibilities and trap options? Or can we hope for a better support of the configuration capabilities of the protocol? Unfortunately, we can be almost sure that configuration will not be more supported than it is today, but that the monitoring possibilities will be greater, i.e a bigger Management Information Base with a lot of managed objects, which is not a bad thing for the network administrator as he uses the monitoring capabilities more than the configuration. It is almost sure that there will never be a better support for the configuration part of SNMP by the vendors. The only possibility can come from applications and the possibility to extend the agents; for example, with *SNMP4J* [15] we have the possibility to write our own agents. If we add some objects in the MIB, it would be rather easy to extend the agent.

The *Simple Network Management Protocol* is obviously *the* standard protocol for network management and it will certainly keep this status in the future, as long as it is widely supported by the most important network device vendors. In our today complex networks, it would be rather difficult to *live* and manage without SNMP.

Operation	**Description**	**SNMPv1**	**SNMPv2**	**SNMPv3**
get	Request initiated by the NMS to get an information from the agent.	**X**	**X**	**X**
get-next	Lets the NMS retrieve a group of values from a MIB by sending a sequence of commands, actually by sending a sequence of *get* operations. The *get-next* operation does a traversal of a subtree, where the beginning node is given in the command.	**X**	**X**	**X**
get-bulk	Permits to get a large section of a table at once. The agent tries to respond with a *get-response* PDU that contains as many objects as it can fit into the PDU.		**X**	**X**
get-response	Operation used by the agent to respond to a *get*, *get-next* or *get-bulk* query.	**X**	**X**	**X**
set	Changes the value of a managed object or creates a new row in a table. Only objects defined with a *read-write* or *write-only* access in the MIB can be modified or created with the *set* command.	**X**	**X**	**X**
trap	A trap is sent by an agent to a NMS when a special event happened, event the agent was configured to catch and signal.	**X**	**X**	**X**
notification	Identical to a trap, but standardizes the PDU format of SNMPv1 traps.		**X**	**X**
inform	Operation that allows manager-to-manager communication. Very useful, if more than one NMS is needed in a network.		**X**	**X**
report	Defined in draft version of SNMPv2, but was never implemented. Is used in SNMPv3 to allow SNMP engines to communicate with each other.		**X**	**X**

Figure 2.5: SNMP operations

Chapter 3

The VeriNeC Project

The VeriNeC project [13] aims to simplify network configuration. It is based upon an abstract definition of a network and the nodes in that network expressed in XML. Each node consists of its hardware (network interfaces) and a set of services such as DNS, packet-filtering, and so on. The abstract configuration is translated automatically into configuration specific to the actual hard- and software used in the network. The simulator part of VeriNeC allows to check if the configuration will fullfill the desired behaviour prior to really configure the nodes. Thanks to VeriNeC, we can greatly improve the security of a network, because when we distribute the different configurations, we know exactly what they do and that they were checked against configuration errors, bad configuration or simply sure not to have a per-default configuration with passwords like 1234 or 0000; at least, we can be sure that the configuration does what we want, and no more.

We will first have a look at the architecture of VeriNeC and see the different modules that compose it. After that, we will discover how the network is described and defined in the abstract definition using the XML syntax. Finally, we will have a closer look at the translation module and its process.
This chapter is just an overview of VeriNeC and focuses only on the parts that will be of importance to understand the next chapter.

3.1 Architecture

In this section, we will have an overview of the architecture of VeriNeC and see the different modules it has. VeriNeC can be divided in four main parts, but has actually many modules as we will see later :

- Simulation and Verification
- Translation
- Importation
- Edition

In Section 3.3, we will focus on the translation part, because this was the part that I had to use for my work; the other parts will only quickly be mentioned (see [13] for more details). As we can see on Figure 3.1[1], VeriNeC is made up by different modules :

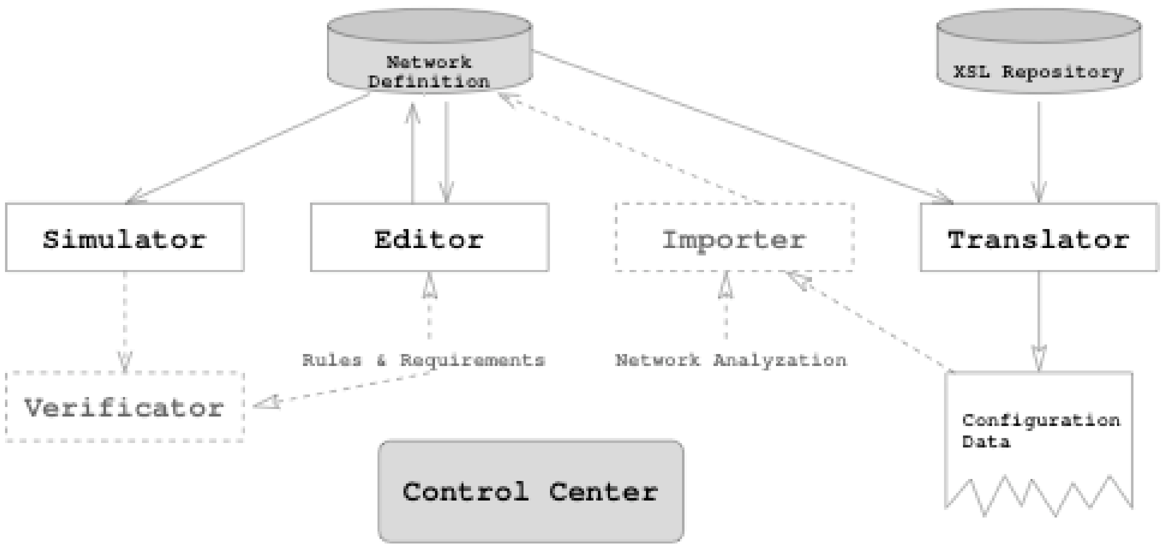


Figure 3.1: VeriNeC's architecture [13]

- **Control center** : graphical application to control every module that runs on the management station
- **Translator** : creates and distributes the configuration data for a given node
- **Simulator** : simulates network behavior from the network definition
- **Verificator** : uses the simulator to test whether or not a network fulfills some given requirements
- **Editor** : displays a network, can modify a node configuration and network layout, and can also show the output of a simulation
- **Importer** : analyses networks and configuration files to create new repository data

You certainly now wonder how all this interconnects together and how this all works, don't you? I still ask for a bit patience, as I need to introduce a few more things before all will become crystal-clear.

3.2 Network Definition

The *network definition* is certainly the central stone, the *clé de voûte* of VeriNeC. As we already mentioned it, the network definition is an abstract definition of a network where every node of the network is described using an XML syntax. A node can represent a router, a hardware firewall, a printer, a workstation, a switch, etc., and every node

[1]Dotted lines denote parts of the system not yet or not fully implemented

contains information on its hardware and services it can provide. In the *hardware* part, we typically specify the interface parameters, and in the *service* one, we can give parameters for routing, packet-filtering, DNS, and so on. For every node, we will also have some information about how to translate this abstract definition into configuration data proper to the node that then can be distributed to it. To understand how this XML abstract definition is organized and how nodes data is stored, we will have a look at the XML schema that describes this.

3.2.1 Network Definition Schema

The XML Schema *nodes.xsd* defines how nodes and their hardware and services are represented. We can see a simple node on Listing 3.1.

Listing 3.1 : A simple node

```
<nodes xmlns="http://diuf.unifr.ch/tns/projects/verinec/node"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://diuf.../verinec/node
                           http://diuf.../verinec/node.xsd">
 <node hostname="macintel06">
  <hardware>
   <ethernet name="EthernetIn">
    <hint system="cisco" slot="Ethernet0"/>
    <ethernet-binding id="bind-16">
     <nw id="nw-75" type="ip" address="192.168.1.33"
       subnet="255.255.255.0" type="ip"/>
    </ethernet-binding>
   </ethernet>
  </hardware>
  <services>
   ...
  </services>
 </node>
   ...
</nodes>
```

The node definition, as we can see it in the above example, has two sections :

1. Hardware
2. Services

The *hardware* element can currently have three different children, i.e interface types :

- ethernet
- wlan
- serial

In Listing 3.1, we can see an ethernet interface with IP address *192.168.1.33* and subnet mask *255.255.255.0* that is bound to network *bind-16*. Ethernet interface can of course only be physically connected to one network, while serial and wlan interfaces can be connected to several different networks. The *services* element contains all the different services that the node provides; currently, the supported services are *routing*, *dns* and *packet-filters*.
As we will see in later examples, the prefix used for the *nodes* elements is usually *vn*, as some other schemas are used in the abstract network definition file for other purposes. If you want more information on the *nodes* schema, you can find it at `http://diuf.unifr.ch/tns/projects/verinec/node.xsd`.

3.3 Translation part

In this section, we will have a closer look at the translation module in VeriNeC. As we already mentioned it, the translation module has to generate configuration data proper to a specific network node from the abstract network definition and then distribute this data to the node using an appropriate distribution method; this can be done with *cp*[2], *scp*[3], *SNMP*, etc. . We can see the translation module in details in Figure 3.2.

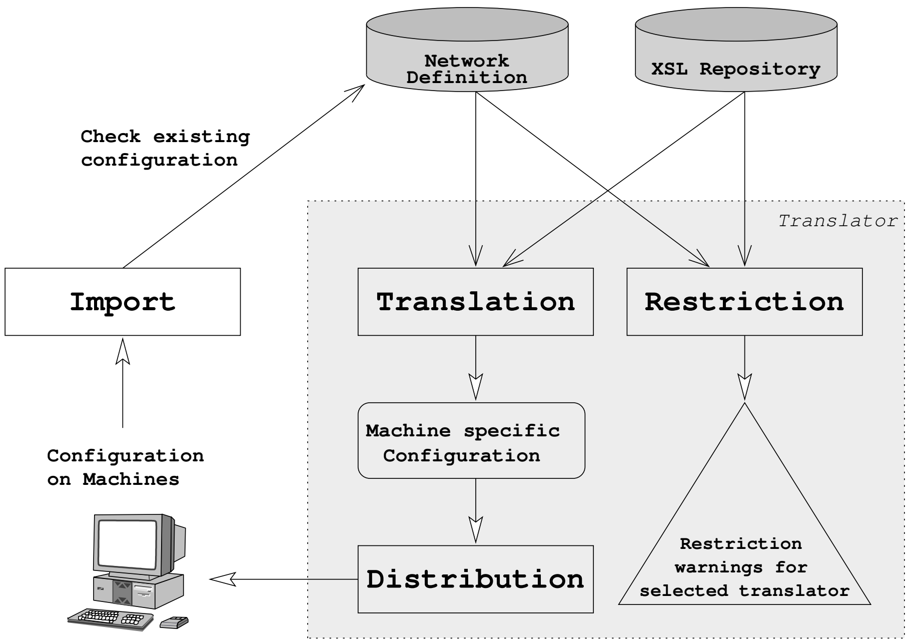

Network Definition
XSL Repository
Check existing configuration
Translator
Import
Translation
Restriction
Machine specific Configuration
Configuration on Machines
Distribution
Restriction warnings for selected translator

Figure 3.2: The translation module [13]

Remember that every node has a hardware section and can provide different services. As a network is a heterogeneous environment, it is rather normal to have different implementations for the same service. If we take for example the packet-filter service, Cisco 800 Series router does it using *access lists*, while Linux uses *ipchains*, *iptables*, *ipfilter* or whatever other implementation. To know which translator the translation module has to

[2]Copy command in Unix and Linux file systems
[3]Same as cp, but adds security using ssh and can connect to other machines.

use for this or that service, we use metadata.
All the translators are stored in the *XSL Repository* (see Figure 3.2) and are implemented using the *eXtensible Stylesheet Language Transformations* (XSLT). This means, as we already discussed it above, that each service has several different translators, i.e XSLT stylesheets, depending on the number of different implementations it has. On Figure 3.3, we can see how the XSL Repository is structured and organized, that a service can have different implementations; it also shows us that the children of the <*hardware*> element, like *ethernet*, *wlan* or *serial*, are in fact treated like the different services.

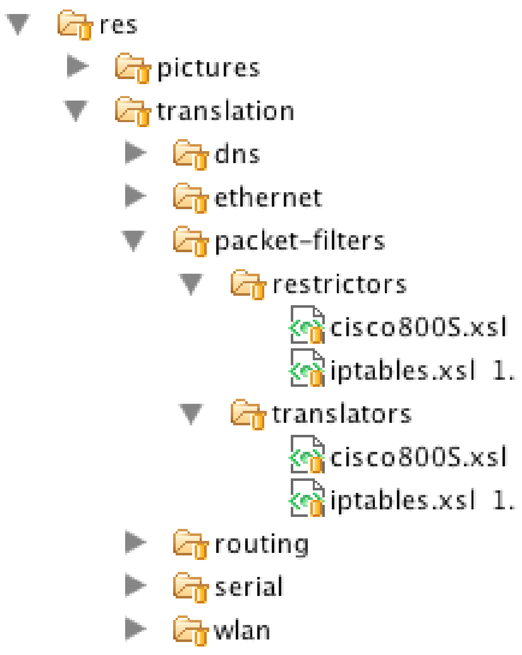


Figure 3.3: XSL Repository structure

The translation module applies the right translator to the given node from the XML abstract network file thanks to metadata (see Figure 3.2), and generates an XML instruction file. This file contains node's system specific configuration data or scripts, gives information to the distribution part on how to connect to the target system and some other information towards specific tasks to be done before and after the configuration has been changed. It can happen that a node parameter cannot be translated, because it is for example not supported (yet) by the target system, so we need some warnings when this situation happens. The solution is to use *restrictors*, that are also XSLT stylesheets, so when a restriction is found, a warning is fired by the restrictor. As we can see in Figure 3.3, every service implementation must have a correspondent restrictor, but it can produce an empty output.
To make the translation possible, we need to put some extra information about it into each node. We need information about the translator, i.e the XSLT stylesheet we need for the translation, and information about the method to connect to the machine to configure. For this purpose, we use a translation schema[4] which defines *types* element to select the translators and *targets* to define the way to connect and send the configuration to the

[4]Its namespace is given by `http://diuf.unifr.ch/tns/projects/verinec/translation`

machine; the namespace used for the translation elements within the XML nodes file is usually *tr*. We will see some examples that show us how this schema can be used.

Listing 3.2 : Global type declaration

```
<nodes xmlns:tr='http://diuf.../verinec/translation.xsd'>
 <tr:typedef def-type-id='ciscodef'>
  <tr:type name='Cisco1' id='cistype01'>
   <tr:service name='ethernet' translation='cisco800S'/>
   <tr:service name='routing' translation='cisco800S'/>
   <tr:service name='packet-filters' translation='cisco800S'/>
  ...
```

In Listing 3.2, we can see an example of a type declaration. It is wrapped in a *typedef* element directly under the root element of the XML network definition. The type in the example declares three services, and for each one, the translator to use for the translation; if we have again a look at Figure 3.3, we can clearly see that for the *packet-filters* service from above Listing 3.2, the translation will look into the *packet-filters* folder of the XSL Repository and choose the one named *cisco800S*. This kind of type declaration, like in Listing 3.2, can then be used as type reference for different nodes, and the *typedef* element also defines a default type that will be used by nodes without type reference.

Listing 3.3 : Type declaration within node element

```
  ...
<node hostname="neochosen1">
 <tr:nodetype type-id="cistype01">
  <tr:service name="ethernet" translation="cisco1200S"/>
 </tr:nodetype>
  ...
```

Nodes can reference types, but can also overwrite some translators. We can see this in Listing 3.3, where the type *cistype01* created in Listing 3.2 is referenced, and where the service ethernet is overwritten. Nodes can create their own types, reference global types and overwrite some translators, or have no *nodetype* element but reference the default type declaration from the *typedef* element (see Listing 3.2).
We have seen so far examples about how types are declared, i.e how the translation process knows which translator to choose from the XSL Repository to create a specific instruction file with configuration data for each node. We will now explain with examples the way to define how the configuration data can be set for the target system to configure.

Listing 3.4 : Targets declaration

```
<node>
 <tr:nodetype>
  <tr:service name="packet-filters" translation="cisco800S">
  <tr:target name="Cisco800S">
   <tr:cisco>
    <tr:snmp targetAddress="10.10.10.1" hostIP="10.10.10.2">
     <tr:security>
      <tr:community snmpversion="v2c">
       private17_veri
      </tr:community>
     </tr:security>
    </tr:snmp>
   </tr:cisco>
  </tr:target>
 </tr:service>
  ...
```

The above listing shows us that the *target* element can be declared in the *nodetype* element. In this example, we can see that the configuration data has to be sent to a Cisco device using a SNMP request; all the parameters that are needed to send the configuration data to the Cisco target system are given. Other systems can use the cp or scp method to send the configuration data[5]. Like the type declaration, targets can also be declared just after the root element tag inside a *service* element of a *typedef* tag and later be referenced from nodes, or be declared outside of a *service* element and be used as default target. This is illustrated with Listing 3.5.

Listing 3.5 : Global target declaration

```
<nodes xmlns:tr='http://diuf.../verinec/translation.xsd'>
 <tr:typedef def-type-id='acgt' def-target-id='deftarget'>
  <tr:type name='typename' id='acgt'>
   <tr:service name='ethernet' translation='linux-fedora'>
    <tr:target name='cpinside-example'>
     <tr:cp prefix='/tmp/inside'/>
    </tr:target>
   </tr:service>
   <tr:service name='wlan' translation='linux-fedora'/>
  </tr:type>
  <tr:target name='tarname' id='deftarget'>
   <tr:cp prefix='/tmp'/>
  </tr:target>
 </tr:typedef>
  ...
```

[5]For more descriptions on the parameters that each method need, have a look at the translation schema you can find at `http://diuf.unifr.ch/tns/projects/verinec/translation.xsd`

In the above listing, we can for example see that the wlan service will use the default target *deftarget* to distribute the configuration data to the Fedora Linux system.

3.3.1 Translation Process

We have seen how the translation module works and what its different components are. We will now have a quick look on the translation process itself and show its different steps :

1. VeriNeC has to determine the type of each node, before the translation can begin. Each node needs to be completely resolved.
2. Warnings are produced if some features are not supported by the target system.
3. The translators resolved during step 1) are applied from the XSL Repository to the node.
4. The configuration data is distributed to the system thanks to the information given by the target elements.

These steps are well illustrated in Figure 3.4.

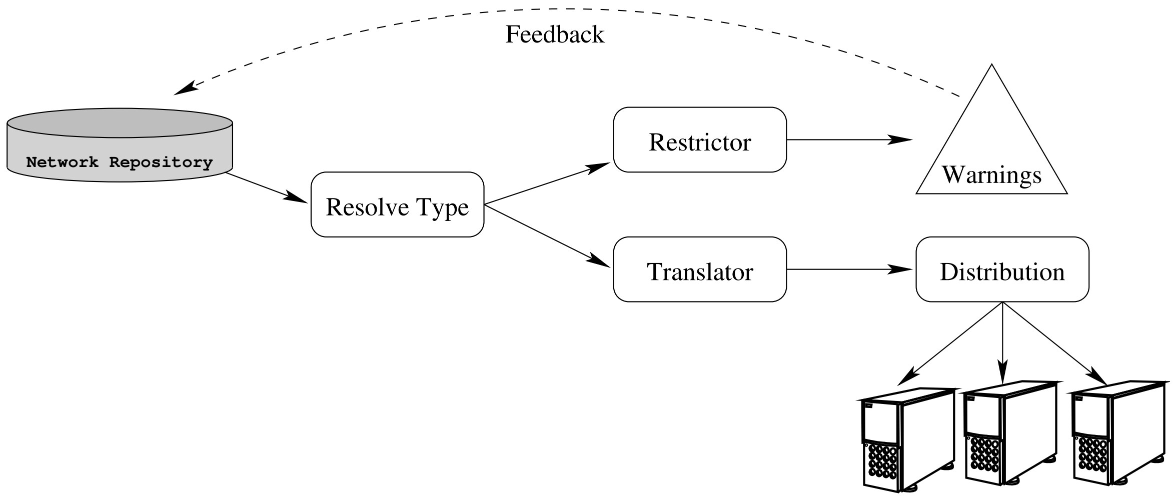


Figure 3.4: Translation process [13]

We have already seen that the translators transform the hardware elements and the services into an XML instruction file with some implementation specific form depending on the target system. If the target system is a Fedora Linux for example and we want to configure its ethernet interface, then the instruction file will contain some configuration data specific to Fedora Linux's ethernet interface. This XML instruction file is conforming the *configuration*[6] schema. Listing 3.6 shows a configuration output for an ethernet interface of Cisco after the ethernet translator has been applied to the node.

[6]Its namespace is given by http://diuf.unifr.ch/tns/projects/verinec/configuration

Listing 3.6 : Configuration output example

```
<configuration
     xmlns="http://diuf.unif.../verinec/configuration"
     xmlns:tr="http://diuf.unif.../verinec/translation">
 <service name='ethernet'>
  <tr:target name='Cisco800S'>
   <tr:cisco>
    <tr:snmp targetAddress='10.10.10.1' hostIP='10.10.10.2'>
     <tr:security>
      <tr:community snmpversion="v2c">private17</tr:community>
     </tr:security>
    </tr:snmp>
   </tr:cisco>
  </tr:target>
  <result-file filename='/tmp/tftp/running-config'>
   Interface Ethernet0
   ip address 10.10.10.1 255.255.255.0
   ip address 192.168.1.1 255.255.255.0 secondary
   ip access-group 122 out
   ip nat inside
   no cdp enable
  </result-file>
 </service>
</configuration>
```

A configuration output is a collection of *service* tags (remember that hardware elements and services are treated in the same way), one for each hardware element and service. The *target* element from the node is also copied into the *service* tag, as this target information will be used for the distribution of the configuration data. The *service* element may also contain some pre- and post-processing commands, and the way how the configuration is outputted (in our example, it is a result-file). For more information about the *configuration* schema, have a look at `http://diuf.unifr.ch/tns/projects/verinec/configuration.xsd`.

Chapter 4

Cisco translation service

We were so far introduced to SNMP in Chapter 2 and to VeriNeC in Chapter 3, both of importance for my project. Chapter 2 revealed the explanation of the first part of the title; this chapter will explain the second part, what will close the loop. As you will see, we will speak a lot about *Cisco Systems* and how I could include it in VeriNeC.
In a first part, I will list the different requirements I had to fulfill in my project; each of the requirements will later be explained in other sections. In a second part, I will present the technology I used to make it possible to solve some problems and to make it possible to achieve the project requirements. After having seen this first two sections, we will attack the implementation part. To make it as interesting and realistic as possible, I decided to follow the different steps I made during the implementation of my project. The first thing I had to do, was to study Cisco; I had to see if it was possible to send a configuration file to a device without using the command line interface, and make myself familiar with the syntax and content of a Cisco configuration file. The next step was to write the translators, and the corresponding restrictors, for the different services I had to support for Cisco. Finally, we will have a look at the different Java classes I had to write. In the last part of this chapter, we will see different points of my Master project that can be improved in further development. Almost every problem we will discuss in this last part will also be mentioned in the concerned sections when this or that problem arises. Some can certainly be quickly solved, while other will need much more work and reflection.

4.1 Project requirements

In this section, we will see the requirements I had to achieve for my project. In simple words, I had to add Cisco support to VeriNeC, what means I had to extend the translation module with a package that provides appropriate translators to create configuration files specific for Cisco, and that extends the distribution with a method to send the configuration to the device.
To accomplish my project, I had some parts to write and others to extend. We will go through them, before we will have a deeper look in Section 4.3 :

- A Java class that implements methods to send and receive Cisco configuration files
- A Java class that implements some utilities for the Cisco configuration

- A Java class that implements the distribution part for Cisco
- A Java classes that implement a TFTP server
- XSLT Translators and Restrictors for the routing, packet-filters and ethernet services
- The Extension of the translation schema for Cisco support
- Study of the Cisco IOS 12.3 and the configuration file syntax

I also had to familiarize myself with the Cisco 831 router I was given and that I used for my project.

4.2 Technology

Before we go into the details of the implementation and the different parts of the project, we will see an overview of the technology I used to realize my project, accompanied by comments why I have chosen it. The technology elements will be divided in two groups : *languages and systems* and *tools and frameworks.*

Languages and systems

Java : I have used and actually had to use Java[1] as programming language, as VeriNeC is completely written in Java. Thanks to this object oriented language, its modularity and simplicity, Java is really a good choice, as everyone can write different parts of VeriNeC, put them into packages and at the end assemble everything together... And yes, it works !

XML : XML cannot really be seen as a programming language, but more as a meta-language. I used XML[21] to write my examples and for testing my translators and implementations. XML is today widely used, because it provides a structured data representation that can be simply parsed and retrieved by different kinds of applications, it is open and it is easy to use. For example, it is used in Mac OS X system preferences file, it describes the structure of a network in VeriNeC, or will even be the base structure of the next Microsoft Word file. XML can today be seen as the central point of a lot of systems and applications.

XSLT : XSLT is a XML-based document language to transform XML files. The XSLT processor uses XSLT rules to transform an XML document into another XML file, or into another format like HTML, plain text, or any other format supported by the processor. This transformation language is widely supported by the community; XSLT is about to upgrade certainly this year to Version 2.0, as the Version 1.0 is presently the recommended one and Version 2.0 is in final working draft status. For VeriNeC, XSLT is important, as it is the language of the translators; we have many different translators for one XML file, in fact the abstract network file. I

[1]If you need more information on Java, please have a look at Sun's Java home page `http://java.sun.com`

used XSLT[23] to write, as already mentioned in Section 4.1, the translators and restrictors for the ethernet, packet-filters and routing services.

XML Schema : XML Schema is also an xml-based language which defines the structure and content of XML documents. *XML Schema* is one of several XML definition languages. We can say that it is the successor of DTD, which also describes the structure of XML documents but used a non-xml syntax. XML Schema was preferred to DTD in VeriNeC, because the first one supports types, among other reasons, and actually no new project would prefer DTD to Schema. I used *XML Schema*[22] to extend the translation schema and to write a SNMP prototype schema (that is not used in my project).

Cisco IOS : Cisco IOS is the operating system used on Cisco Systems routers, firewalls and switches. I had to study the Cisco IOS system, its commands and the syntax of the configuration files that are in fact IOS commands, as we will see in Section 4.3.1.2. We will speak more in details about Cisco in Appendix A.

Tools and frameworks

Eclipse : I used the free Eclipse[2] IDE platform for the whole development, including the XSLT stylesheets and XML Schemas. We can also add some powerful and very useful plugins to Eclipse that at the end will result in an all-in-one development tool.

oXygen : oXygen[3] is a powerful XML editor written in Java, with a rather cheap education license. This XML editor can be used both as a standalone application or as an Eclipse plugin; this application has a lot of features, among them a nice and simple to use XML editor, a visual Schema editor or a XSLT debugger. I used the Eclipse plugin a lot to write my XSLT translators, test them, extend the Schemas or write some XML examples and test files.

SNMP4J : This is a SNMP implementation in Java, presently the best and most advanced I have found. SMNP4J[15] supports SNMPv1, v2c and v3, what is not the case for almost all other Java SNMP implementations.

JUnit : This is a powerful testing framework that permits to write simply and quickly unit tests in Java. Testing is actually a very important and sometimes underestimated task in a development cycle, as it permits to find some problems and bugs in the code during development; the more tests we write, the most confident we can be about our code. I used JUnit[4] to test my functions and classes, and it permitted me to find some errors I had not expected.

Now that we have seen the technology part, it is about time to switch to the implementation details.

[2]`http://www.eclipse.org`
[3]`http://www.oxygenxml.com`
[4]Have a look at `http://www.junit.org` or `http://junit.sourceforge.net/`

4.3 Implementation

For this implementation section, we will discuss the three different blocks I had to work with to carry my Cisco translation module off. Whenever possible, we will see some code snippets that illustrate the implementation we are looking at that moment. I will also explain the reason why I implemented this or that code snippet like this and not in another way.

We will first see the secrets of Cisco IOS and its configuration files in Section 4.3.1. After that we will speak about the translators in Section 4.3.2, each one supporting a service, and the XML Schema I had to extend to include my part of the project. Finally, we will see in Section 4.3.3 the different classes I implemented to make the distribution to the Cisco device possible.

4.3.1 Cisco part

The first thing I had to see was how it is possible to configure the Cisco 831 router, or more generally Cisco devices, in a suitable way for VeriNeC; this means in a way that we can configure remotely the device, without having to connect manually to it, in a way that a configuration can be distributed to the router. Once I found a way to do it, I had to focus on the syntax of the configuration file and the architecture of Cisco IOS, and write in parallel the translators.
We will in a first part concentrate on the way we can distribute the configuration on the device, and in a second part we will discuss the syntax of Cisco's configuration file.

4.3.1.1 Distribution method

Paradoxically, the first thing I had to look for was the possibility to transfer the configuration to Cisco devices in a way suitable for VeriNeC, in fact the last step in the translation process (see 3.3). If there was no such possibility, my project goals could not be achieved; this is the reason why I began indeed at the end of the translation process.
After browsing around on Cisco's huge website[5], I finally found an interesting technote having the title *How To Copy Configurations To and From Cisco Devices Using SNMP*. You now certainly react like I reacted : "Fantastic, it is at least possible to configure most Cisco devices with SNMP". Unfortunately, as we saw it in Section 2.6, SNMP is not used as it is intended to. What we actually do is send an SNMP command (see Listing 4.2) to the device that tells it to download a given configuration file (see Section 4.3.1.2) at a given address using a given protocol. The MIB Cisco recommends to use for copying configurations is called CISCO-CONFIG-COPY and is supported on the 831 router since IOS version 12.3-11.T3[6]. In Listing 4.1, we can see the object that shows how to use the CISCO-CONFIG-COPY MIB :

[5]It is sometimes difficult to directly find what you are looking for, since the website structure is not always good.

[6]On older versions of the IOS, other MIBs could be used, like the *CISCO-FLASH* or *OLD-CISCO-SYSTEM*, but they disappear in later versions or are not recommended by Cisco, and anyway are not as powerful as the *CISCO-CONFIG-COPY*.

Listing 4.1 : Part of CISCO-CONFIG-COPY MIB

```
CcCopyEntry ::=
   SEQUENCE {
      ccCopyIndex                         Unsigned32,
      -- configuration items
      ccCopyProtocol                      ConfigCopyProtocol,
      ccCopySourceFileType                ConfigFileType,
      ccCopyDestFileType                  ConfigFileType,
      ccCopyServerAddress                 IpAddress,
      ccCopyFileName                      DisplayString,
      ccCopyUserName                      DisplayString,
      ccCopyUserPassword                  DisplayString,
      ccCopyNotificationOnCompletion      TruthValue,
      -- status items
      ccCopyState                         ConfigCopyState,
      ccCopyTimeStarted                   TimeStamp,
      ccCopyTimeCompleted                 TimeStamp,
      ccCopyFailCause                     ConfigCopyFailCause,
      ccCopyEntryRowStatus                RowStatus
        }
```

It is important to understand that each creation of such a *config-copy request* object creates an entry in a table, so that we can track the status of the object and delete it from the table when we do not use the object anymore; the entry number in the table is given in the SNMP command (see Listing 4.2). The object we can see on the above listing is a sequence of different objects, which we will briefly explain. The first object, of type *Unsigned32*, specifies the index of the object in the table. The second object specifies which protocol, default one is *tftp*, we use to copy the configuration file to or from the router; the *ConfigCopyProtocol*[7] type defines five different supported protocols, where each one is accessed by the integer between parentheses :

- tftp (1)
- ftp (2)
- rcp (3)
- scp (4)
- sftp (5)

If ftp, rcp, scp or sftp is used, we have to specify a username string for the *ccCopyUserName* object, and if ftp, scp or sftp is used, we have to specify a password string for the *ccCopyUserPassword*. The third and fourth object specify the types of files on which a config-copy operation can be performed; like with the protocol specification, each file type is selected by its integer between parentheses :

[7]All the special types are defined in the same MIB; have a look at the CISCO-CONFIG-COPY for more details.

- networkFile (1)
- iosFile (2)
- startupConfig (3)
- runningConfig (4)
- terminal (5)

Either the *ccCopySourceFileType* or the *ccCopyDestFileType* (or both) must be of type *runningConfig* or *startupConfig*. The next two objects specifies the IP address of the server where the router can download or upload the configuration and the name of the file on the server. The last object of the configuration items, by default set to false, specifies whether or not a *ccCopyCompletion* notification should be issued on completion of the tftp transfer; we have to be sure that notifications are allowed to be delivered.
The first object of the status items gives the actual state of the copy operation; the *ConfigCopyState* defines different states :

- waiting (1)
- running (2)
- successful (3)
- failed (4)

If state 4 is returned, i.e the copy request failed, we can ask the *ccCopyFailCause* object for the cause of the failure. The *ccCopyTimeStarted* object specifies the time the *ccCopyState* last changed to the running state, or 0 if the state has never changed to running (what can happen when stuck in waiting state), and the *ccCopyTimeCompleted* object specifies the time when the *ccCopyState* last changed from the running state to successful or failed state. Finally, the last object of the status items is the status of the table entry; it can be set to activate, so that the specified table entry will be processed, or it can be set to delete status so that the table entry will be deleted and freed.
Now that we have seen the *CcCopyEntry* object and its sequence of objects, we will be able to understand easily the large SNMP command we have to use for a config-copy request :

Listing 4.2 : SNMP config-copy request

```
snmpset -v 2c -c private17_veri 10.10.10.1
  1.3.6.1.4.1.9.9.96.1.1.1.1.2.333 i 1
  1.3.6.1.4.1.9.9.96.1.1.1.1.3.333 i 1
  1.3.6.1.4.1.9.9.96.1.1.1.1.4.333 i 3
  1.3.6.1.4.1.9.9.96.1.1.1.1.5.333 a 10.10.10.2
  1.3.6.1.4.1.9.9.96.1.1.1.1.6.333 s running-config
  1.3.6.1.4.1.9.9.96.1.1.1.1.14.333 i 4
```

First we can see on Listing 4.2 above that every object ends with the number 333; this is the entry table number, and we can choose the number between 1 and 999, but before using the same entry again, we have to be sure that it is free. The first object reference, ending with 2.333, tells us that we will use the tftp protocol to copy the file, as it has selected the integer (i) number 1 (see above for the integer numbers of the other available protocols). The second object reference, ending with 3.333, specifies that we will use a *networkFile* as source file type, corresponding to the integer 1, and the third object, ending with 4.333, specifies that we use *startupConfig* as destination file type, corresponding to the integer 3. The fourth object reference specifies the IP address (a) of the tftp server, that is in our case 10.10.10.2 . The fifth reference gives the name of the file on the server to use, in our case *running-config.* The last object reference, ending with 14.333, makes that the command is processed immediately by setting the integer 4. If you have not understood yet what this command does, it copies the file called *running-config* from the tftp server with address 10.10.10.2 to the router using tftp protocol.
Once the command is executed, we can follow the status of the copy request, using the *ccCopyState* object we have seen above

Listing 4.3 : SNMP check status request

```
snmpwalk -v 2c -c private17_veri 10.10.10.1
  1.3.6.1.4.1.9.9.96.1.1.1.1.10
```

The command in Listing 4.3 will give us an integer as output, as we already saw it above; as long as the copy request is working, we will receive 2 as output, but as soon as the operation has ended, we will receive hopefully state 3 or, if something went wrong, state 4. When our request ended successfully, it is better not to forget to delete the entry of the table, so that this entry can be used again immediately by another request; this can be done using the *ccCopyEntryRowStatus* object and setting it to integer value 6 :

Listing 4.4 : Deleting the entry table

```
snmpset -v 2c -c private17_veri 10.10.10.1
  1.3.6.1.4.1.9.9.96.1.1.1.1.14.333 i 6
```

Now we have seen how it is possible to copy a configuration file to and from a Cisco device, we will have a look at the configuration file itself.

4.3.1.2 Configuration file

Cisco IOS has two main configuration files where you can find every feature and parameter that are configured for the device :

- startup-config
- running-config

The *startup-config* file is read during the startup process of the device, the file that is the per-default or initial configuration of the device. The *running-config* file reflects the actual configuration and parameters of the device, what means that just after the startup

process has ended, the *startup-config* file and the *running-config* files are identical. If we modify the configuration or add some new features to the device using CLI, this will modify the *running-config* file, but not the *startup-config* file, which is actually a kind of backup : if we misconfigured something and are stuck, we can simply reset the device which will use again the *startup-config* file. If we are sure that our modifications work fine and that it is what we wanted, we also have to change the *startup-config*, what can be done by copying the content of the *running-config* to the *startup-config* file. This is exactly what we can do using the command we saw in last Section 4.3.1.1, where we can specify as file type *startupConfig* or *runningConfig*.
It is also important to notice, that we do not need to send the complete configuration file to a device each time we have made modifications. It is enough to just send the portion of the configuration that was modified, but sometimes we need to send some extra commands with it, like for example if we want to delete or modify a rule in an access list; we will see this more in details in Section 4.3.2.2.

We have now to have a look at the content of such a configuration file itself; in Listing 4.5 we can see parts of it.

Listing 4.5 : Parts of a configuration file

```
...
! Ethernet port 0
interface Ethernet0
 ip address 10.10.10.1 255.255.255.0
 ip access-group 122 out
 ip nat inside
 no cdp enable
 hold-queue 32 in
!
! Access-list example
access-list 111 permit tcp any any eq telnet
access-list 111 permit icmp any any echo
access-list 111 permit icmp any any traceroute
access-list 111 deny   ip any any
...
```

We will not go into the details of Listing 4.5[8], but if we look closely at it, we see that each line corresponds to a command we would have typed if we had used the command line interface (CLI) to configure the device. For example, to configure Ethernet port 0, we first have to type[9] *interface Ethernet0* before we can configure it; after that, we can assign it an IP address (10.10.10.1) and a subnet mask (255.255.255.0) with the command *ip address 10.10.10.1 255.255.255.0*, etc. We can see, that if we want to understand a configuration file, we have to understand the different CLI commands we would use to configure the whole device. For the purposes of my project, I did not need to know every

[8]Have a look at the *reference guides* of Cisco IOS on Cisco's homepage for more information about each command.

[9]We will see more details about Cisco configuration in Appendix A

command or understand the whole content of a configuration file; I had to focus on the commands that were useful for my needs. I had to find the commands that deal with the services I had to support in my work : ethernet, packet-filters and routing. Sometimes, for packet-filtering and routing for example, there exists different commands to do it with sometimes more or less parameters and features, so I had to make choices and select the one that suited me best. We will speak again in Section 4.3.2 about the commands I used for the different services when I will present the different XSLT stylesheets for each service and the strategies I adopted.

4.3.2 Translators and schema extensions

We have now everything in the hands to write the translators (see Chapter 3) for the different services that have to be supported in my project :

- ethernet
- packet-filters
- routing

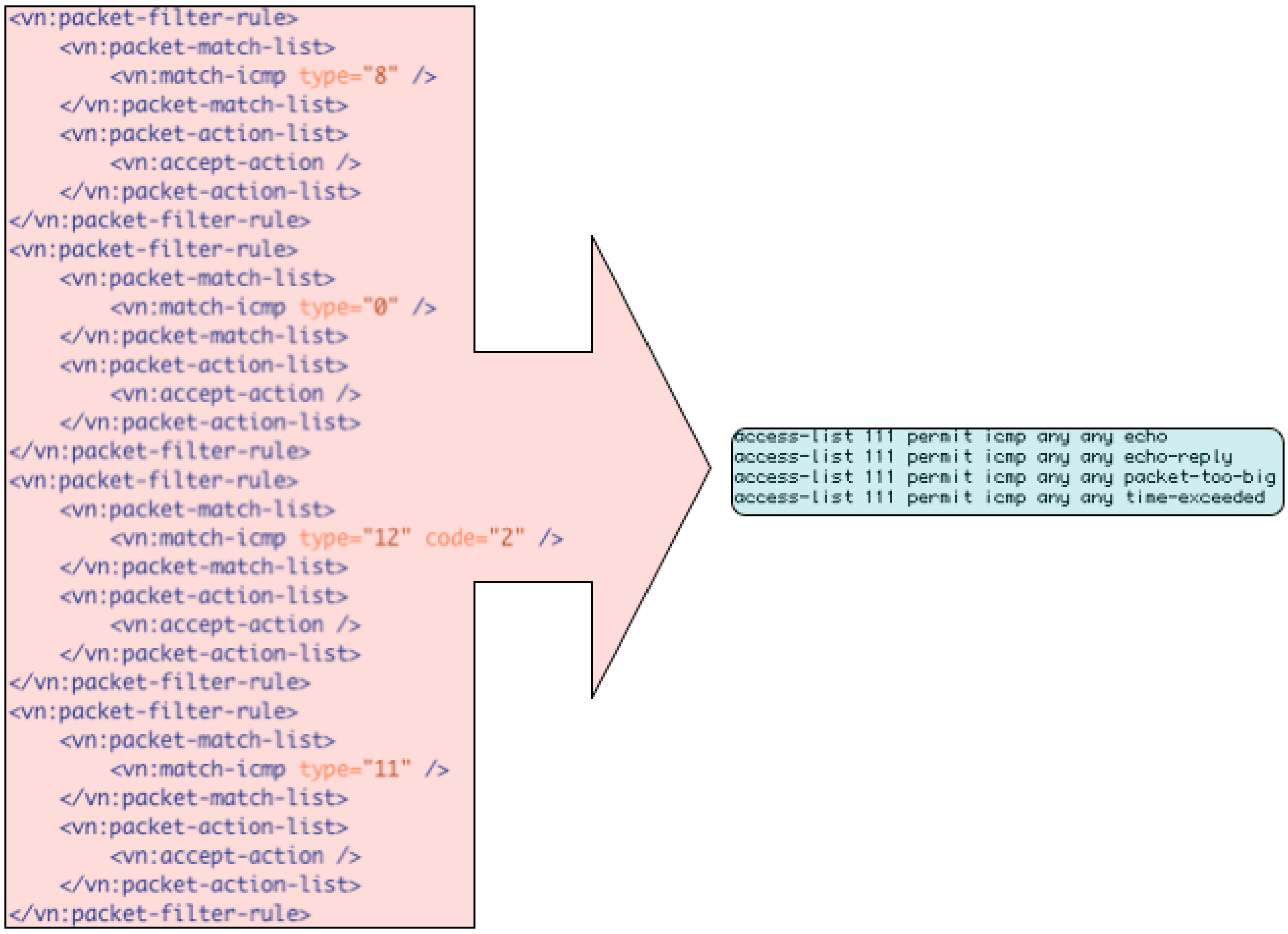
We have seen in Section 4.3.1.2 the syntax and the format of a Cisco configuration file, so as we now know the result we have to find after the translation, we can write the translators for each service that will translate the abstract configuration document into a Cisco configuration file; we can see a little example of the result on Figure 4.1 on Page 36. We will speak about each translator and come again on the IOS commands, introduced in previous section, that are used to reflect the abstract configuration. We will see some concrete examples with the *nodes* XML file that illustrate best the translation from one state to another. We will of course not go through the whole XSLT file, but only speak about the most important parts of it. We will also see that sometimes it is not possible to translate a configuration exactly like it should be, and that sometimes we should need more information from the abstract configuration in order to configure the Cisco device properly. This aspects will also be discussed in the next three sections.

4.3.2.1 Ethernet translator

The translator for the ethernet service was certainly not the easiest to write, because I had to deal with parameters I should have needed for the Cisco configuration file but that were nowhere to be found in the abstract configuration document, or information that were in the *ethernet* element of the abstract configuration, but located somewhere else in the Cisco configuration.
Listing 4.6 shows us a typical example of how the ethernet interfaces are configured with Cisco, so we can see what should be the result of our translator.

Listing 4.6 : Ethernet interface Cisco configuration

```
...
interface Ethernet0
 ip address 10.10.10.1 255.255.255.0
 ip access-group 122 out
 ip nat inside
 no cdp enable
 hold-queue 32 in
!
interface Ethernet1
 ip address dhcp client-id Ethernet1
 ip access-group 111 in
 ip nat outside
 ip inspect myfw out
 duplex auto
 no cdp enable
 ...
```



```
<vn:packet-filter-rule>
    <vn:packet-match-list>
        <vn:match-icmp type="8" />
    </vn:packet-match-list>
    <vn:packet-action-list>
        <vn:accept-action />
    </vn:packet-action-list>
</vn:packet-filter-rule>
<vn:packet-filter-rule>
    <vn:packet-match-list>
        <vn:match-icmp type="0" />
    </vn:packet-match-list>
    <vn:packet-action-list>
        <vn:accept-action />
    </vn:packet-action-list>
</vn:packet-filter-rule>
<vn:packet-filter-rule>
    <vn:packet-match-list>
        <vn:match-icmp type="12" code="2" />
    </vn:packet-match-list>
    <vn:packet-action-list>
        <vn:accept-action />
    </vn:packet-action-list>
</vn:packet-filter-rule>
<vn:packet-filter-rule>
    <vn:packet-match-list>
        <vn:match-icmp type="11" />
    </vn:packet-match-list>
    <vn:packet-action-list>
        <vn:accept-action />
    </vn:packet-action-list>
</vn:packet-filter-rule>
```

```
access-list 111 permit icmp any any echo
access-list 111 permit icmp any any echo-reply
access-list 111 permit icmp any any packet-too-big
access-list 111 permit icmp any any time-exceeded
```

Figure 4.1: Abstract configuration to Cisco configuration

As we can see it in Listing 4.6, we have generally at least two Ethernet interfaces configured, one connected to the inside network (Ethernet0), and one to the outside network (Ethernet1).
The first important parameter we have to find in the abstract configuration under the *ethernet* section is the use of the <*hint/*> element; with this element, we specify that the

ethernet configuration is for a Cisco device[10] and we use the *slot* attribute of the element to specify which interface we want to configure. For interface *Ethernet0* from Listing 4.6, we would have the example we can see on Listing 4.7 .

Listing 4.7 : Use of the <hint/> element for Cisco

```
...
<ethernet name="EthernetIn">
  <hint system="cisco" slot="Ethernet0"/>
...
```

If this element is not used, it is impossible for the system to know the name of the ethernet interface, and if other attributes are used within <*hint/*>, the restrictor would complain and explain what is not right.

The next element we will introduce is the use of <*nw-dnsserver*>, the element that deals with DNS server information. In the *node* XML Schema, we can see that this element is attached to an interface and each interface can have as many DNS servers as needed. If we now have a look at Cisco IOS documentation, we first see that DNS server information is not bound to a specific interface and thus declared outside of their configuration; moreover, we can only specify six DNS servers. The Cisco IOS command that permits to define name servers is *ip name-server* followed by the IP address(es) of the name servers and is to be used in global configuration mode[11]. The translator scans all the *ethernet* elements in the abstract configuration to look if there is a <*nw-dnsserver*> element and writes it in the output configuration; if ten are found, ten will be written, but when the file is distributed to the Cisco device, it will only take into account the first six one. The problem is that there is no way to specify which ones are the most important DNS servers to use, if there are more than six. Of course, if there is no reason to give one DNS server more importance than to another, we could simply add a recursion to the translator that would only write the first six found DNS servers into the output configuration.

Another difference between the abstract VeriNeC and Cisco configuration is the way packet-filter rules are assigned to interfaces. Within the *nodes* XML Schema, we can see that we can assign a *packet-filter-chain* to an interface in the <packet-filters> element, as this is done in the interface configuration with Cisco using the *ip access-group* command. This last command refers to an *access-list* (see Chapter 4.3.2.2) using its number, the one that follows *access-group*. With the translator, once I am about to write the output configuration, I have to jump to the packet-filters section to see if there is an <*if-map*> element in the <*interface-filter-mappings*> element attached to the current processed <*ethernet*> element; I can find this thanks to the *interface* attribute that references the *id* attribute of its <*ethernet-binding*>. Once I found an <*if-map*> element, I have to jump to the <*packet-filter-chain*> element referenced by the *chain* attribute, where I will finally find the number of the *access-list* to write in the output configuration file.

There are still some problems with the translator that could not be resolved due to

[10]The *hint* tag is not only used for Cisco, but also for several other systems.

[11]Have a look at Appendix A for more information about the different configuration modes.

parameters proper to Cisco and impossible to express within the abstract configuration. It is impossible to activate the *Cisco discovery protocol* (expressed with *cdp* in Listing 4.6), disabled per default, with VeriNeC. There is also no possibility to parameter *nat*; this problem was partially solved using hard code, choosing that *Ethernet0* will normally have *ip nat inside*, and *Ethernet1 ip nat outside*, but it is not possible to change this using VeriNeC, as there is no element that deals with *NAT*. For this special cases, and there are certainly more I have not seen yet, we should perhaps create an extra XML Schema which would permit to define all this extra parameters, but then we would have a problem with the simulation part. The last problem I had with my 831 router, was more a hardware problem; it is impossible to really configure the ethernet hub ports from the router, like giving an IP address or a netmask, because these interfaces are working on layer 2.

As some parameters from the abstract configuration document are not supported for Cisco, or at least for the 800 series routers, I had to write a restrictor for the ethernet service. For more details about the restrictor, I invite you to have a look at it in the repository.

4.3.2.2 Packet-filters translator

The translator for the *packet-filters* service was certainly the one that cost me most reading and searching. I had to find what corresponds as good as possible to the filtering rules definition from the abstract file, as with Cisco, you have different possibilities to filter packets, even dynamic packet filtering. The chosen solution was also the closest one to the packet-filter-chain of the abstract definition. I used the *access-list* IOS command as solution, which you can see as example on Listing 4.8.

Listing 4.8 : *access-list* example

```
access-list 111 permit tcp any any eq telnet
access-list 111 permit icmp any any administratively-prohibited
access-list 111 permit icmp any any echo
access-list 111 permit icmp any any echo-reply
access-list 111 permit icmp any any traceroute
access-list 111 permit icmp any any unreachable
access-list 111 permit udp any eq bootps any eq bootpc
access-list 111 permit udp any eq bootps any eq bootps
access-list 111 permit udp any eq domain any
access-list 111 permit udp any any eq isakmp
access-list 111 permit udp any any eq 10000
access-list 111 permit tcp any any eq 139
access-list 111 permit udp any any eq netbios-ns
access-list 111 deny   ip any any
```

With above listing, we can see that the *access-list* command works like the *<packet-filter-chain>* element :

- a number (111 in the example) that permits to know to which access-list (chain) a rule belongs to

- a *permit* or *deny* action
- a pattern definition, i.e filtering parameters (Figure 4.3 for more details)
- order importance of the rules

The number identifying the access-list does not only play the identification role, but also serves to know to which type of access-list it belongs. See Table 4.2 for an example of the access list number ranges that we can use to filter traffic :

<1-99>	IP standard access list
<100-199>	IP extended access list
<200-299>	Protocol type-code access list
<300-399>	DECnet access list
<600-699>	Appletalk access list
<700-799>	48-bit MAC address access list
<800-899>	IPX standard access list
<900-999>	IPX extended access list
<1000-1099>	IPX SAP access list
<1100-1199>	Extended 48-bit MAC address access list
<1200-1299>	IPX summary address access list
<1300-1999>	IP standard access list (expanded range)
<2000-2699>	IP extended access list (expanded range)

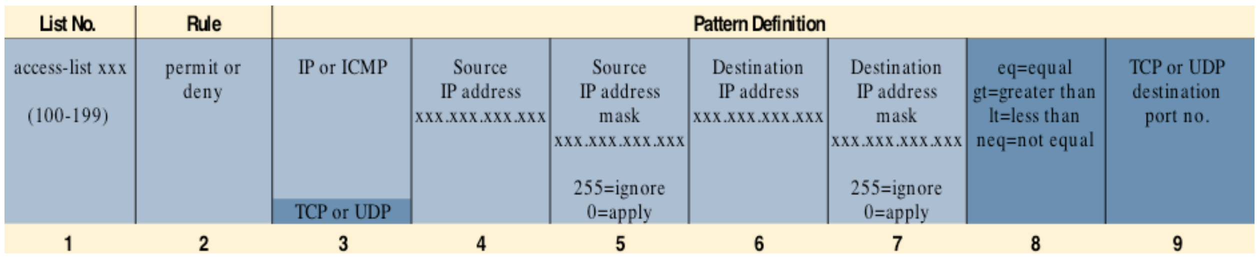
Figure 4.2: Access list number ranges [2]

For my project, I only had to use the *IP extended access list*, which corresponds to ranges numbers <100-199> and <2000-2699>. As the number has an importance and is what identifies the access list ”membership”, I had to find a way to use it in the abstract definition. I decided to use the *name* attribute of the <packet-filter-chain> element to put the important access list number, as we can see it on Listing 4.9.

Listing 4.9 : Using ACL number

```
...
<packet-filter-chain name="111" id="pfc2">
...
```

The problems I had were less with the ACL numbers, but more with the way how the IOS configures the access lists or takes the changes into account. You cannot simply upload a new file to the Cisco device with just the rule to modify or the rule to delete (using the *no access-list* command). You have to first erase the entire access list rules, for example with *no access-list 111* to delete the access-list from Listing 4.8, and rewrite again the whole access list with the rule to modify or without the rule you want to delete. As I do not know if the abstract configuration for the access list is the same as the one already on the device, I have each time to delete first all the access lists and rewrite everything from what is given in the abstract configuration. An important thing I do not know yet, is whether or not the fact that all access lists are first deleted could open a security hole for a very short moment. This depends on the way the device reacts when configuration parameters are changed. We will see some propositions to resolve this in Section 4.4.



List No.	Rule	Pattern Definition						
access-list xxx (100-199)	permit or deny	IP or ICMP / TCP or UDP	Source IP address xxx.xxx.xxx.xxx	Source IP address mask xxx.xxx.xxx.xxx 255=ignore 0=apply	Destination IP address xxx.xxx.xxx.xxx	Destination IP address mask xxx.xxx.xxx.xxx 255=ignore 0=apply	eq=equal gt=greater than lt=less than neq=not equal	TCP or UDP destination port no.
1	2	3	4	5	6	7	8	9

Figure 4.3: Access list syntax (fields in dark blue are optional) [25]

Like with the ethernet translator, I also had to write a restrictor for the packet-filters service. Some options of the abstract configuration are not supported for Cisco, but there are actually also a lot of parameters in the access lists that are not possible, for the moment, to express in the abstract configuration. If you want more information on the restrictor, I invite you to have a look at it.

4.3.2.3 Routing translator

The routing translator was certainly the easiest and shortest one to write. I only had to support static routing, leaving dynamic routing support for future improvements. With Cisco IOS, you have a lot of different commands and possibilities for doing routing, but the simplest one and also the closest to the <*static-routes*> element parameters, is to use the *ip route* command (in global configuration mode). This command has quite a lot of argument possibilities, but I will only present a simplified version of it that is sufficient for our understanding and needs. We can see this simplified version of the *ip route* command on Listing 4.10 .

Listing 4.10 : Simplified *ip route* command

```
ip route prefix mask {ip-address} [distance]
```

For example, *ip route 10.1.0.0 255.0.0.0 171.28.2.3 110* tells that packets for network 10.1.0.0 will be routed through a router at 171.28.2.3 if dynamic information with administrative distance less than 110 is not available. The (optional) distance number can have a value between 0 and 255; the smaller the number is, the better we can trust the routing information. Every routing protocol has a default distance[12], for example *static route* has value 1, IGRP 100, OSPF 110, RIP 120, etc. .
If we look at the <static-routes> element example from the abstract configuration on Listing 4.11, we can already see a few problems I had to solve within the translator or using a restrictor.

Listing 4.11 : <static-routes> element example

```
<static-routes>
 <route metric="22" scope="global" zone-index="as2">
  <route-destination address="192.168.7.3" length="24"/>
  <nexthop via="10.10.1.23"/>
```

[12]Have a look at the node XML Schema under routing section for a complete list

```
  <nexthop device="i80_3"/>
  <nexthop via="1.2.3.4"/>
  <nexthop weight="123"/>
 </route>
</static-routes>
```

If we have a look at the attributes of the <*route*> element, we can see that there is no support for them in Cisco IOS, what means that a restrictor is at least needed for them. If you have a look at the restrictor, you will see that I had to take some other parameters in consideration.
Next, we see that the netmask value, given with the *length* attribute of the <*route-destination*> element, is not in dotted notation like it has to be in Cisco configuration, but in integer notation where the number corresponds to the number of bits (from left) that have 1 as value; so 24 = 11111111.11111111.11111111.00000000 (255.255.255.0[13]). I decided to do the conversion from the integer notation to the dotted notation within the translator, as so I will not need to do it later, which would cost to process again the whole configuration file just for doing the conversion.
If we compare the <*static-routes*> element definition with the *ip route* command, we can see that the former can have as many <*nexthop*> elements as it wants and having different parameter possibilities, while the latter can have only one next hop information and this information can only be the IP address of this next hop. All the <*nexthop*> elements with another attribute than *via* will be detected by the restrictor, and if there are more than one with the *via* attribute, only the first one will be taken as next hop; the rest will be discarded and the restrictor will give a warning to the user.
The last problem I had with the routing translator was testing the result; I could not see if my routing settings worked or not, as the device was not connected to a network and this seems to be the condition to make the *ip route* parameters to be written in the Cisco configuration file.

4.3.2.4 Translation XML Schema extension

The only XML Schema I had to extend was the *translation* schema. I added a <*cisco*> element that contains only an <*snmp*> element; if we have a look at it in Listing 4.12, we could say that the <*cisco*> element has no reason to exist and is superfluous, since using the <*snmp*> element from Listing 4.13 as first element would have the same result.

Listing 4.12 : <cisco> element definition

```
<xs:element name="cisco">
 <xs:complexType>
  <xs:sequence>
   <xs:element ref="snmp" />
  </xs:sequence>
 </xs:complexType>
</xs:element>
```

[13]I think that it would be much easier to use a *netmask* attribute value instead of the *length* attribute. This could perhaps be changed in the *node* XML Schema.

I decided to make it like that to show that SNMP will be used for Cisco and not in a way we would think it would be used (have again a look at Section 2.6 if this sentence is not clear to you). Next, we need a way to give all the needed options for the SNMP command, like the SNMP version we want to use, the IP address of the device and the TFTP server[14], and the different security parameters needed for the different versions of SNMP.
In Listing 4.13, we define the <*snmp*> element :

Listing 4.13 : <snmp> element definition

```
<xs:element name="snmp">
 <xs:complexType>
  <xs:sequence>
   <xs:element ref="security" />
  </xs:sequence>
  <xs:attribute name="targetAddress"
                type="IPv4" use="required" />
  <xs:attribute name="hostIP" type="IPv4" use="required" />
 </xs:complexType>
</xs:element>
```

It is in this element that we have to specify the IP address of both the Cisco device and TFTP server. Next, comes the definition of the <*security*> element :

Listing 4.14 : <security> element definition

```
<xs:element name="security">
 <xs:complexType>
  <xs:choice>
   <xs:element ref="community" />
   <xs:element ref="snmpv3" />
  </xs:choice>
 </xs:complexType>
</xs:element>
```

If you remember what we have seen in Chapter 2, SNMP has three different versions, but two different used security models. If we have a look at Listing 4.14, we can see that in the <security> element, we have to choose between the community based security model used by SNMPv1 and SNMPv2c, and the security model for SNMPv3. In Listing 4.15, we can see the definition of the first security model :

Listing 4.15 : <community> element definition

```
<xs:element name="community">
 <xs:complexType >
  <xs:simpleContent>
   <xs:extension base="xs:string">
```

[14]The server needs to run on the same machine VeriNeC is running on, but this can be changed with a little bit of coding.

```
    <xs:attribute name="snmpversion" use="required">
     <xs:simpleType>
      <xs:restriction base="xs:string">
       <xs:pattern value="v1|v2c" />
      </xs:restriction>
     </xs:simpleType>
    </xs:attribute>
   </xs:extension>
  </xs:simpleContent>
 </xs:complexType>
</xs:element>
```

In this <community> element, we have to specify of course the community name, but also to precise which of SNMPv1 and v2c has to be used. If we now come to the next security model, we are already sure that SNMPv3 is used. This security model, as we have already seen it, is much more secure and complete :

Listing 4.16 : <snmpv3> element definition

```
<xs:element name="snmpv3">
 <xs:complexType>
  <xs:sequence>
   <xs:element name="username" type="xs:string"/>
   <xs:element name="securityLevel">
    <xs:simpleType>
     <xs:restriction base="xs:string">
      <xs:pattern value="NOAUTH_NOPRIV |
                         AUTH_NOPRIV | AUTH_PRIV"/>
     </xs:restriction>
    </xs:simpleType>
   </xs:element>
   <xs:element ref="auth" minOccurs="0"/>
   <xs:element ref="privacy" minOccurs="0"/>
  </xs:sequence>
 </xs:complexType>
</xs:element>
```

Listing 4.16 reminds us that with this security model, we have support for authentication and privacy. It is so not surprising to find an element where we have to specify the *username*; if we only give the username, we are using *NOAUTH_NOPRIV* as *securityLevel*, what would actually be the same as the community based security model. If we use *AUTH_NOPRIV* as *securityLevel* value, we have to add the <auth> element, and with *AUTH_PRIV* we have to use both <*auth*> and <*privacy*> elements. In Listing 4.17, we can see the definition of the <auth> element :

Listing 4.17 : <auth> element definition

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

```
  <xs:attribute name="function" use="required">
   <xs:simpleType>
    <xs:restriction base="xs:string">
     <xs:pattern value="MD5|SHA"/>
    </xs:restriction>
   </xs:simpleType>
  </xs:attribute>
  <xs:attribute name="passwd" type="xs:string" use="required"/>
 </xs:complexType>
</xs:element>
```

In this element, we just have to choose which authentication function we want to use between *MD5* or *SHA*, and of course give the password that will be used by the function. Finally, we can see in Listing 4.18 the definition of the *<privacy>* element :

Listing 4.18 : <privacy> element definition

```
<xs:element name="privacy">
 <xs:complexType>
  <xs:attribute name="function" use="required">
   <xs:simpleType>
    <xs:restriction base="xs:string">
     <xs:pattern value="AES128|AES192|AES256|DES"/>
    </xs:restriction>
   </xs:simpleType>
  </xs:attribute>
  <xs:attribute name="passwd" type="xs:string" use="required"/>
 </xs:complexType>
</xs:element>
```

Like with the authentication element, we have to specify which encryption method we want to use between *DES* and different flavors of *AES*, and of course give the password used for encryption/decryption.

These different definitions we have just seen can certainly also be used in the future, if one day VeriNeC wants to add a true SNMP distribution service. There should perhaps just be an *OID* element to add. Before this becomes reality, we have to wait on a better support from the vendors.

4.3.3 Java

The Java implementation part is the last one we still have to discuss. If we have a look at VeriNeC's translation module, we will see that the only Java class I have to write is a distributor for my Cisco service that implements the *verinec.translation.IDistributor* interface; this is the interface that every distributor has to implement. As you can imagine, this was certainly not the only class I wrote for my project, as I needed much more functions and procedures to be able to distribute the configuration file to the Cisco 831 router. We will see each of the classes one by one, explaining what they do, why I had to implement this or that, what special library I used. I will of course not explain every function of each class or explain the code in details, because if you are interested in this, I think that it is better to read the Java documentation or directly have a look into the code.

4.3.3.1 TFTP Server

You certainly remember our discussion of Section 4.3.1.1 on Page 30 about the distribution method for Cisco. We saw that there are different protocols that can be used to exchange a file with Cisco devices :

- TFTP
- FTP
- RCP
- SCP
- SFTP

I decided to implement the TFTP protocol, because it is the one that is apparently most used for such tasks, the most cited one in Cisco's tech notes, and also the easiest one to implement. TFTP stands for Trivial File Transfer Protocol and is a very basic form of FTP; it has no authentication nor encryption support, cannot list directory contents and works on UDP port 69. If you want to write a file to a TFTP server, you have to be sure that there already exists a file, empty or not, with that file name. As this protocol is generally used in local or private networks, we do not have to be concerned by the lack of security, unless we are in a very big corporate network. Since it uses UDP, it is up to the application to have a kind of acknowledgment mechanism.

The implementation of TFTP was straightforward, as the protocol is described in only one RFC, namely RFC 1350, and that I just had to follow this RFC. Fortunately, I found an already complete implementation of TFTP on the exercises website of the Information Technology group of the Rochester Institute of Technology[15]. I modified the different classes for my needs, mostly getting rid of the graphical user interface part. This TFTP implementation consists of four classes we will now see one by one.

TFTPPacket
This class contains all that has to do with a TFTP packet, like the different opcode and error values, the different fields of a packet, the functions to get and set the values of these fields, and of course a method to parse and one to build a packet. All the values and the format of a packet are all directly taken form TFTP's RFC.

Download
The download class defines all the different fields and methods needed for the download process from the TFTP server to a client. It also uses a timer to resend lost packets.

[15]This little TFTP implementation seems to have no license restriction.

Upload

If there is a download class, there is of course also an upload one. This class contains all the fields and methods to upload a file to the TFTP server from a client.

TFTPServer

Finally, we have the class that implements the TFTP server. This class contains everything that is needed to do a nice server, to start and stop the service.

4.3.3.2 CiscoTransmit

The *CiscoTransmit* class contains all the methods to send and receive a configuration file to/from a Cisco device using SNMP to start the file transfer with TFTP. Since we exactly know the different OIDs we will use with their corresponding values in the SNMP command, we can process everything that has to do with SNMP in this class. As I already mentioned it, I used SNMP4J[15] to write the code where SNMP was needed. SNMP4J is quite simple to use and has a lot of features that I did not need to use. I implemented the send and receive methods for SNMPv1 and v2c, but let SNMPv3 support for a future update. At the beginning, I decided to add this when everything will work fine with SNMPv1 and v2c, but because of a lack of time, I had to abandon the idea. Nevertheless, I have already created the different variables and the constructor for having SNMPv3 support; the only two methods that still have to be implemented are the sending and receiving of a configuration file. SNMP4J uses also a different type of PDU[16] for SNMPv3 as for the two other versions, reason why we have to write the methods for it in a different way. We also first need to configure the Cisco device for SNMPv3 support, as we have to add some parameters before this can work.

4.3.3.3 CiscoUtil

CiscoUtil is a very little class that contains only one method for the moment. This class should in the future be put in the *verinec.util* package, as it is intended to contain a collection of various methods for Cisco configuration file. For example, at present the only method is used when access lists are to be distributed to the device; the method parses the configuration searching the *access-list* commands and each time a new access list number is found, a *no access-list ACL_number* command is added at the beginning of the configuration file[17]. I am sure you remember the problems I alluded to in Section 4.3.2.2; one of these problems was the way we have to take to delete or modify an access list rule. If you remember, we first have to delete all the access lists and then write them again without the rule we want to delete or with the rule we have modified. The method we mentioned above is here to be sure that all the access lists will be deleted. This method is used by the *DistCisco* class for access lists distribution just before the configuration file is send to the Cisco device.

[16]You will find the meaning of PDU in Appendix B.

[17]This could be done with XSLT, but I already thought at the situation when there will be a Cisco Importer module which will give us the possibility to compare the rules, and in this case we will need a Java class.

4.3.3.4 DistCisco

DistCisco is my distributor and is the class that implements the *IDistributor* interface that every distributor has to implement. Every distributor has three methods to implement :

- setTarget
- distribute
- execute

The first method, *setTarget*, initializes a distributor with a given target, which is the method argument; in our case the child of this target is a <*cisco*> element. Once we have this element, we can extract all the different information we need for the distribution, like the IP addresses of the host and the server, the name of the configuration file, which SNMP version is used, etc. .
The *distribute* method, which takes a *config* element as argument, performs the actual distribution work. The first thing I have to check, is that the right sort of configuration data is used, namely the <*result-file*> element, which is the *config* element name. After that, I create two files in the TFTP server folder : one that will contain the configuration content to distribute to the device and one temporary file that already contains the configuration commands which is the content of the *config* element. Before writing the configuration command to the right file, I have to check if I have to send access lists configuration part to the device; if this is the case, I have first to use the method we have seen in the *CiscoUtil* class, which will parse the content of the temporary file and write its output in the configuration file that will be sent. Finally, we have to write the configuration commands in the right file, send this file to the Cisco device using the methods from the *CiscoTransmit* class, and delete the temporary file. Now you can understand, why I have chosen *result-file* as sort of configuration data and not another one.
Finally, the *execute* method just returns *Null* as I do not have to execute a special command for the target.

In this part, I had some problems[18] with the *LocalSaxBuilder* class from the *verinec.util* package. Even if it was told to make the Schema validations using the local ones, it still wanted to make a connection to the Internet. As I could not be connected to the Internet while the router was connected to my computer, I could never load my XML configuration tests in *VeriNeC Studio*[19], as I always received an error because it could not validate the schemas. In order to make my testings possible, I had to comment out some lines to disable schema validation in the *LocalSaxBuilder* constructor; in Listing 4.19, you can see which are these commented lines :

Listing 4.19 : Modified constructor of *LocalSaxBuilder*

```
private LocalSAXBuilder() throws VerinecException {
  super("org.apache.xerces.parsers.SAXParser");
  //setFeature ("http://xml.org/sax/features/validation", true);
  //setValidation(true);
  String schemas = buildSchemaLocations();
```

[18]Meanwhile, the following problem description has been fixed.
[19]The graphical user interface of VeriNeC

```
    setProperty("http://apache.org/xml/properties/
    schema/external-schemaLocation", schemas);
    setProperty("http://apache.org/xml/properties/
    schema/external-noNamespaceSchemaLocation", schemas);

    //setFeature ("http://apache.org/xml/features/
    //validation/schema", true);
    setFeature ("http://apache.org/xml/features/
    validation/schema-full-checking",true);
  }
```

In Listing 4.19, we can see that I put all the lines in comments that have to do with Schema validation. This is not a good solution, since there is no schema validation anymore and what could lead to wrong configurations, but it was the only solution for me to do my testings.

4.4 Improvements and critics

With every chapter, we have unveiled new parts of my Master thesis, we have seen the different problems I had to solve, but we could also read the difficulties that appeared, sometimes from nowhere, and that still have to be solved in a future development. In this section, I want to resume the different points of my project that has to be improved or changed in a future release.

4.4.1 SNMPv3 support

As we have seen it in Section 4.3.3.2, my project has unfortunately no support for SNMPv3, because of a lack of time. Even if I almost implemented everything to have a support for this version, we still have to write the send and receive methods in the *CiscoTransmit* class, and also configure the Cisco device with the different parameters that are needed for SNMPv3. Even if the SNMP protocol is mostly used in a local or private network, we still have the risk that someone from the inside of the network, typically in a big enterprise network, is watching the traffic and could see the community name and the different configuration parameters we are sending with SNMP. This is the reason why we need SNMPv3, since this version supports authentication and privacy using cryptographic functions. As security becomes today a critic factor for every enterprise, we cannot take the risk to propose a service or an application that sends important information on the network without any security features.

4.4.2 TFTP alternative

In section 4.3.3.1 we discussed why I have chosen TFTP as protocol for sending or receiving a configuration file, despite the fact I could have some better choice. Unfortunately Cisco cannot specify another port than standard UDP port 69 for TFTP. I say unfortunately, because every port number that is smaller than 1024 needs administrator rights to be

used. We need to launch the server as root or administrator, what is not really ideal for us. The biggest problem is that we cannot start the server with VeriNeC, what is actually something we really want to avoid. This needs to be changed in a future development by using another protocol to transfer the configuration files.
TFTP has also a weakness we just discussed in Section 4.4.1 : lack of security. With this protocol, you have neither authentication nor privacy support. Even if we would have SNMPv3 support, we could still capture the whole content of the configuration file once it is sent with TFTP. What we need is a protocol that will not need some special privileges to be used and has support for authentication and privacy. I would suggest to use SFTP or better SCP, as the latter has meantime been added to VeriNeC. Changing the protocol to transfer the configuration file to/from the Cisco device will be very simple, if the chosen protocol has already been integrated in VeriNeC. We just have to change the value of the concerned OID to tell the device to use SCP instead of TFTP, and add the OID where I have to give the password that will be used by SCP or SFTP.

4.4.3 Cisco importer

When we spoke about access lists and the problem bound to it (Section 4.3.2.2), we mentioned that an Importer module in VeriNeC that would import configurations from Cisco would be very useful. This would be very interesting for comparing the actual configuration from the abstract file with the imported configuration from the Cisco device. We would so for example not need to delete first all the access lists rules and write them all again in order to modify or delete one rule; we just had to compare the access lists entries, find the differences, delete only the access list entry that was really modified, and then distribute only this access list entry and not everything.
A Cisco importer module would also be very useful to import the actual configuration of the device, when we add it for the first time in the abstract configuration. With this arises a new question: what have we to do with all the parameters, commands and options that fit nowhere into the abstract configuration document? This will certainly be one of the most important problem the Cisco Importer module developer will have to solve. I think that a Cisco Importer is a project worth doing.

4.4.4 Testing network for VeriNeC

You certainly remember Section 4.3.2.3 on Page 40 where we have discussed the routing translator. At the end of this section, I said that I was not able to configure the Cisco device with my routing parameters, because it could not find the next-hop IP address since I am not connected to any network. It would be good to have the possibility to test our different configurations on a testing network just for VeriNeC. We will certainly find some bugs we can see only once we are testing our configurations on a network.

Chapter 5

Conclusion

My Master thesis was focused on the network management theme; we first discussed the possibilities of the standard management protocol SNMP, but we had to admit that it is not well implemented in network devices, and then we introduced VeriNeC and showed how it was possible to implement a Cisco translation service in it. The fact that my initial Master thesis, called *SNMP Research*, was not feasible has finally two consequences of importance : the first one is that it shows us the way how network devices vendors (under)use the **S**imple **N**etwork **M**anagement **P**rotocol, and the second consequence is that we now have a first integration of Cisco Systems devices into VeriNeC.

We first began in the second chapter to see what SNMP is, how it works, what are its components. We have also shown how this standard management protocol is integrated into vendors' network devices; they implement it in their system in a way that suits them best : the vendors only integrate the monitoring capability of SNMP, leaving the configuration part aside. With this strategy, they can create a rather lucrative business by giving or selling their own solutions to configure their devices and proposing courses and certifications to master their products. Once a user is accustomed to a system, he will not easily change to another network devices vendor. The fact that vendors do not give the possibility to configure their devices with SNMP is firstly to avoid the customers to be able to easily change to a competitor, as with SNMP we can disregard their complex system and only concentrate on their MIBs[1]; secondly, and this is a direct implication of the first argument, this is a way to guarantee the fidelity of new and regular customers. The lack of the configuration ability in vendors' SNMP implementation is the direct cause why my initial Master thesis was not feasible.
Was indeed the word *deception* in my Master thesis title really too strong toward SNMP? The word *disappointment* would have been a possible replacement, but I think that it does not really traduce the way how the vendors use SNMP, even if most people feel more disappointed than really deceived by this management protocol. When I say that vendors do not use SNMP correctly, this is not completely right, since they implement the monitoring ability in a correct way, but I actually want to raise the attention on the fact that they underuse or even misuse SNMP. We can compare this situation to an hypothetical bank that gives their customers the possibility to open an account, to put some money on it, to check the balance, but does not give them the possibility to withdraw

[1]Remember that the **M**anagement **I**nformation **B**ase uses a standard syntax to describe its managed objects; so, even if the systems are different, the MIB syntax stays the same.

some money from their account.
The future of SNMP depends mostly on the network devices vendors as it is especially them which use it. If we put the business factor aside, we can ask why it is not more supported. Does this protocol not reflect anymore today's managers needs? Should we try to develop a new management protocol? The fact that SNMPv3 has become standard version since 2004 will perhaps give more impact to this protocol, as this standard version supports authentication and privacy using cryptographic functions. If this protocol continues to be used by the network devices vendors in the same way than today, we should perhaps think to change the meaning of SNMP into "Simple Network Monitoring Protocol".

In the third chapter we saw what VeriNeC is, how it works, what are its different modules and what they do, and what is its core; we finally focused on the translation module, since my Master project had to extend it with a Cisco translation service. In today's growing number of heterogeneous interconnected devices in a network, it is absolutely essential that every device is well configured, otherwise this can open security holes or create some other network problems. VeriNeC is designed to solve these problems by giving the possibility to simulate the behavior of a device with this or that configuration, what permits to find configuration problems before we send the configuration to the device. VeriNeC is designed in a way that permits to integrate any kind of network devices, supporting different distribution methods. The secret and core of VeriNeC is the abstract definition of a network where each device or node is expressed in XML; each node has its own configuration parameters and uses different translators to convert the abstract configuration into a configuration specific to the device. This abstract description document permits VeriNeC to disregard the underlying system of the described device.
Verinec improves network security and avoids bad configurations thanks to the simulator module. The translation and importer modules open VeriNeC the door to almost every network device we can imagine. VeriNeC will provide the network administrator a transparent tool to manage all the devices of his network; he will be able to configure a Cisco router or a Hewlett-Packard printer without having to know how Cisco IOS works or how he has to connect to the HP printer to configure it. VeriNeC is a precursor of a new kind of network management tools.

The fourth chapter showed the different parts of my Master thesis, the problems I encountered to achieve the requirements and the different strategies I used to solve them. In the first two sections, we could see the different technology elements I used and the project requirements I had to achieve. The next section explained us the implementation part of my Master thesis. We first saw how it was possible to distribute a Cisco IOS configuration file in a suitable way for VeriNeC and interested us in the syntax and content of such a configuration file. We then saw the translators for the different services I had to support and the XML Schema extension I had to do to add my distribution method. The last part of the implementation section presented the different Java classes I implemented to distribute a configuration to a Cisco device. In the last section of this fourth chapter, we enumerated the improvements that could be done in a future development.
With this chapter we can really have an excellent insight of how the translation process of VeriNeC works and above all how we can extend VeriNeC with our own translation service for this or that network device. In our case, we extended VeriNeC with a Cisco translation

service, what is indeed really a good thing because if we think at the infrastructure that is used in big enterprises networks, we almost always find some Cisco Systems devices. If VeriNeC wants one day to be used in such big networks, it really has to support at least Cisco routers, firewalls and switches. This observation leads us to a problem I had while implementing the translators and this is actually a problem we will have with almost every device : how can we integrate specific important commands or options of a device into VeriNeC? Cisco IOS has for example so many commands and options which cannot be translated into the abstract definition document. This can show a kind of chosen limit of VeriNeC : on one side we can choose to give the developers the possibility to add their XML Schema for their devices to take these special commands or options into account; on the other hand we can decide what is really needed as configuration on every device to guarantee security and that it provides the different services we can expect from a network device. The first scenario allows us to include all these special options into the abstract definition document, but with several drawbacks : the simulation of the configuration will become almost impossible and using more options could raise the risk to open security holes. The second scenario is the one that was chosen by VeriNeC and also seems to be the better one, as we can simulate every configuration for every network device and we can guarantee that the system does not become too heavy.

My Master thesis is the first work I could find that gives a critical opinion on the way SNMP is implemented in network devices; it clearly shows that SNMP is misused by vendors by not implementing the configuration capability, what permits them to create a new business. It is also a new proof that shows once more the will of the vendors to not follow standards, as they do not want to depend on standards organization, preferring proprietary solutions to make more money with.
My project also implements the first integration of Cisco Systems devices into VeriNeC. Since their devices are presently the most used in enterprises networks, this opens VeriNeC the doors to a new horizon. If VeriNeC wants to have a great impact in the community, it is essential that it supports Cisco Systems devices. I hope that my Master thesis represents the beginning to an important integration of Cisco devices into VeriNeC and that this integration will be continued with future projects.

Appendix A

Cisco Systems

In this chapter, we will speak a bit more about Cisco Systems and what are the first steps to do when we receive a Cisco device and want to make it work with VeriNeC. We will also have a quick look at Cisco IOS command modes structure, each mode permitting to configure a number of parameters.

A.1 A brief history

In 1984, the married couple *Leonard Bosack* and *Sandra Lerner*, who worked in different computer departments at Stanford University, founded cisco Systems (notice the small *c*). Strong with their own experience to solve their trouble getting their individual systems to communicate, they founded cisco Systems with a small commercial gateway server product that changed networking forever. The first product on the market was called the Advanced Gateway Server (AGS). While cisco was not the first company to sell routers, it created the first commercially successful multi-protocol router that permitted previously incompatible computers to communicate using different network protocols.

You now certainly wonder why I always wrote *cisco* with a small *c*. The name *cisco*, which is not an acronym, is an abbreviation of *San Fran**cisco***; the founders had the idea of the name and logo when they saw the *Golden Gate Bridge*[1] while driving to San Francisco. Having the Golden Gate Bridge in mind, have a look at Cisco's logo on Figure A.1.

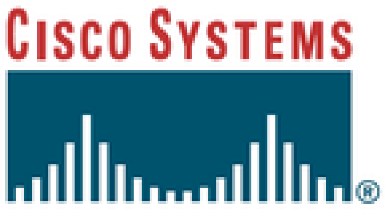


Figure A.1: Logo of Cisco

[1]The Golden Gate Bridge is a well-known bridge in San Francisco, certainly the most famous suspension bridge of the world.

I am pretty sure that you clearly recognized the Golden Gate Bridge. In 1992, the company name was changed to *Cisco Systems, Inc.* .

Cisco has now a big list of hardware and software network products, among them we can find routers, switches, firewalls, Wireless Access Points, IP Phones, etc. . Presently, Cisco is clearly one of the most important vendor of network devices and solutions[2].

A.2 First steps

In this section, we will see what we have to do when we receive a brand new Cisco device and want to make it work with VeriNeC. I will also explain the problems I encountered during the initial configuration.
We will go step by step through the different parameters to configure, even if this method seems perhaps a bit childlike, but it is for me the best way to be sure that everything will work, and this is also intended to be used as reference when new Cisco devices have to be added to VeriNeC.

1. The first step we have to do, is to configure the basic parameters of the device using, if possible, the *Cisco Router Web Setup Tool* (CRWS); this setup tool provides a graphical user interface for configuring easily and quickly Cisco SOHO series and 8oo series routers. You can configure parameters like NAT, DNS or port filtering, but the only things we need, is to parameter the name and the password of the device, and DHCP.
   For this first step, I already encountered the first problem. CRWS is an applet that we can load using a browser. The problem, is that presently this applet does not work with the JRE of Sun, but only with the one of Windows what implies that finally we can only use CRWS with Microsoft's browser, i.e *Internet Explorer.* Cisco is aware of the problem for a long time, but has not upgraded the tool yet.

2. The second step we have to do, is to upgrade the IOS of the device. To be able to download the newest IOS for your device from Cisco's website, be sure to have the client number to be able to access the restricted area where you will find what you are looking for. Once you have downloaded the image of the IOS, you need to upload it to the device; be careful to have enough space left on the device to achieve the upload operation. There are different methods to install a new IOS image on a device, even one using SNMP and TFTP; the best thing you can do is to have a look at Cisco's website and choose the one that suits you best.

3. Next we have to connect to the router using the *telnet* command with the IP address of the router (per default 10.10.10.1, but have a look at the users guide of the device to be sure). Once we are connected, we need to enter privileged EXEC mode (more about this in next section) to password protect this mode, which has per default no password. You just need to type the command *enable secret Your_password.* Next time you will access to the privileged EXEC mode, you will need to enter the password you have chosen. You can easily recognize that you are in the privileged EXEC mode, because each prompt line on CLI ends with the character #.

[2]This is my point of view and has nothing to do with publicity for Cisco Systems !

4. As we are already in privileged EXEC mode since last step, we can now continue and enable the SNMP server. If the server is not running on the device, we cannot send any SNMP request to it and we cannot distribute any configuration. First, we have to enter the configuration mode, then we have to enable the read-only (RO) and read-write (RW) community strings, exit out of the configuration mode and finally write the modified configuration to nonvolatile RAM (NVRAM) to save the settings. We can see all the different commands to enable the SNMP server in Listing A.1.

Listing A.1 : Enabling the SNMP server

```
Router#configure terminal
Enter configuration commands, one per line.  End
with CNTL/Z.
Router(config)#snmp-server community public RO
Router(config)#snmp-server community private RW
Router(config)#exit
Router#write memory
Building configuration...
[OK]
Router#
```

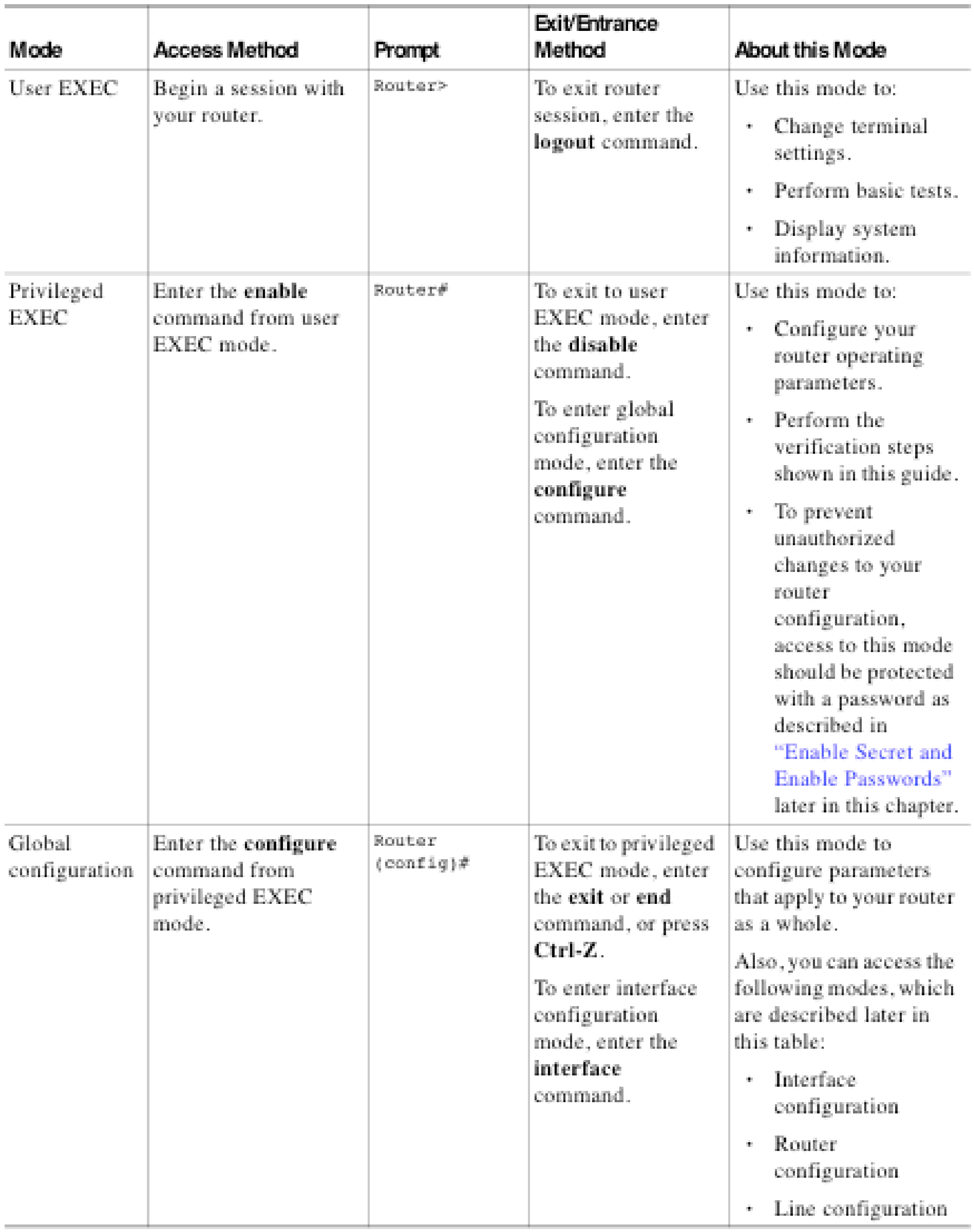
Once all these steps have been performed, we can be sure that the Cisco device will work with VeriNeC. If it does not work properly, make sure you have done all the steps in the right order. After that, if there is still a problem, try to find which of the steps is problematic and look at Cisco's website to find some documentation about your problem or explain your problem on Cisco's forums.

A.3 Command modes structure

In this section, we will see the different Cisco IOS command modes structure. Each command mode supports specific commands; for example, we saw in Section A.2 that the *enable secret* command has to be used in privileged EXEC mode, while we needed to be in global configuration mode to enable and configure the SNMP server. The command modes are hierarchical, what means that we have to go through different modes before we can access the one we want. We have the following hierarchy, from bottom to top :

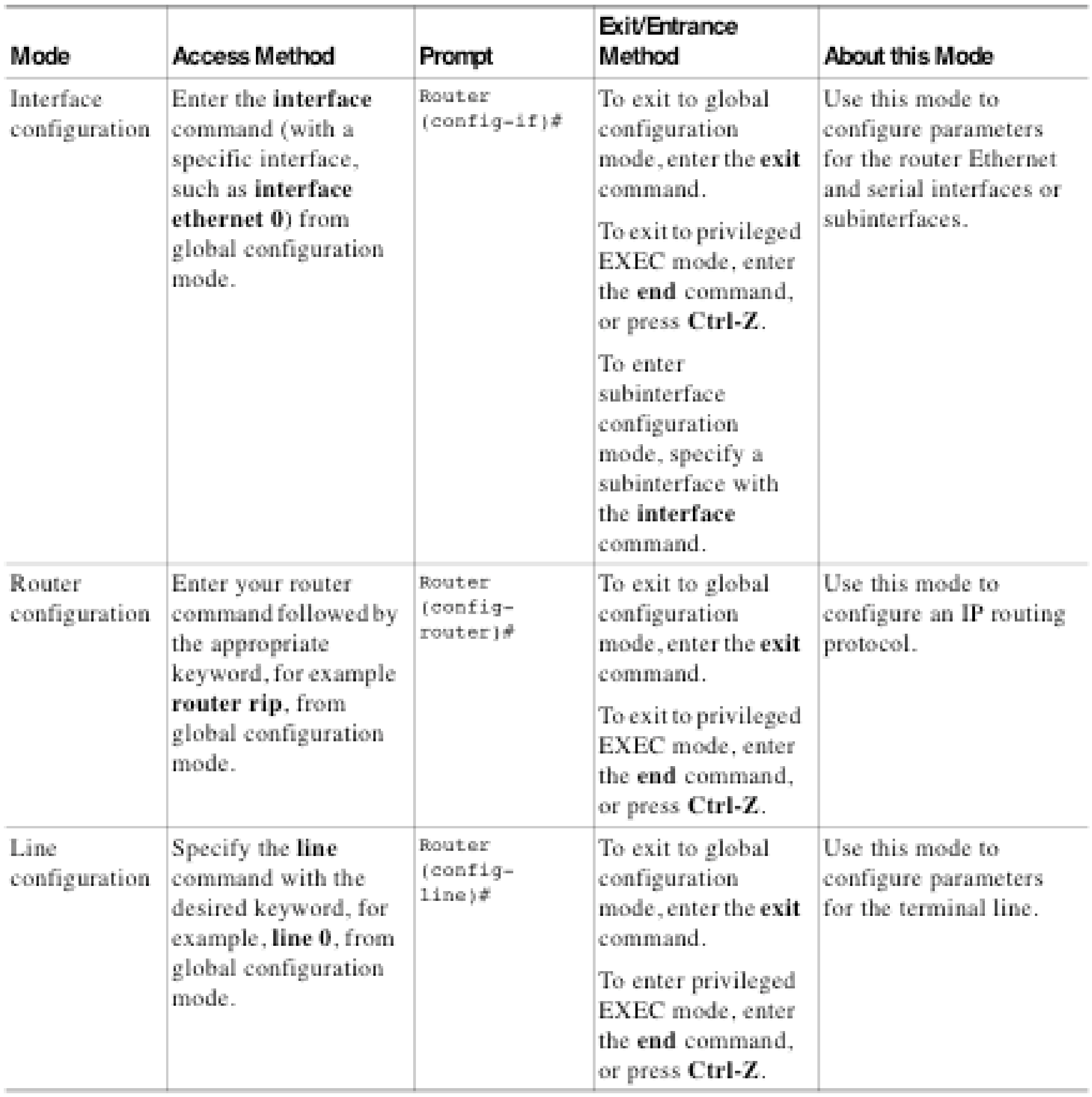
- User EXEC
- Privileged EXEC
- Global configuration

When we begin a session, we are in user EXEC mode. From the global configuration mode, as we will see on Figure A.2, we can reach three other modes, each one permitting to configure a certain element of the device. Figures A.2 and A.3 will show us how we can reach each mode, what are their prompt, how we can exit or enter a new mode, and what we can do when we are in this or that mode.



Mode	Access Method	Prompt	Exit/Entrance Method	About this Mode
User EXEC	Begin a session with your router.	`Router>`	To exit router session, enter the **logout** command.	Use this mode to: • Change terminal settings. • Perform basic tests. • Display system information.
Privileged EXEC	Enter the **enable** command from user EXEC mode.	`Router#`	To exit to user EXEC mode, enter the **disable** command. To enter global configuration mode, enter the **configure** command.	Use this mode to: • Configure your router operating parameters. • Perform the verification steps shown in this guide. • To prevent unauthorized changes to your router configuration, access to this mode should be protected with a password as described in "Enable Secret and Enable Passwords" later in this chapter.
Global configuration	Enter the **configure** command from privileged EXEC mode.	`Router (config)#`	To exit to privileged EXEC mode, enter the **exit** or **end** command, or press **Ctrl-Z**. To enter interface configuration mode, enter the **interface** command.	Use this mode to configure parameters that apply to your router as a whole. Also, you can access the following modes, which are described later in this table: • Interface configuration • Router configuration • Line configuration

Figure A.2: Command modes summary[4]



Mode	Access Method	Prompt	Exit/Entrance Method	About this Mode
Interface configuration	Enter the **interface** command (with a specific interface, such as **interface ethernet 0**) from global configuration mode.	`Router (config-if)#`	To exit to global configuration mode, enter the **exit** command. To exit to privileged EXEC mode, enter the **end** command, or press **Ctrl-Z**. To enter subinterface configuration mode, specify a subinterface with the **interface** command.	Use this mode to configure parameters for the router Ethernet and serial interfaces or subinterfaces.
Router configuration	Enter your router command followed by the appropriate keyword, for example **router rip**, from global configuration mode.	`Router (config-router)#`	To exit to global configuration mode, enter the **exit** command. To exit to privileged EXEC mode, enter the **end** command, or press **Ctrl-Z**.	Use this mode to configure an IP routing protocol.
Line configuration	Specify the **line** command with the desired keyword, for example, **line 0**, from global configuration mode.	`Router (config-line)#`	To exit to global configuration mode, enter the **exit** command. To enter privileged EXEC mode, enter the **end** command, or press **Ctrl-Z**.	Use this mode to configure parameters for the terminal line.

Figure A.3: Command modes summary (continued)[4]

Appendix B

Acronyms

AES Advanced Encryption Standard is in cryptography a block cipher adopted as an encryption standard by the US government.

ASN.1 Abstract Syntax Notation one is a standard notation that describes data structures for representing, encoding/decoding and transmitting data.

BER Basic Encoding Rules are ASN.1 encoding rules.

CLI The Command Line Interface is a way to interact with a computer by giving it textual commands.

DES Data Encryption Standard is in cryptography an encryption algorithm which was the predecessor encryption standard of AES since 1976.

IANA The *Internet Assigned Numbers Authority* is an organisation that supervises IP addresses, domain names and Internet protocol numbers assignment.

IETF The Internet Engineering Task Force is an organization that is responsible to develop and promote Internet standards.

IOS Internetwork Operating System is the operating system used on most Cisco Systems devices.

MIB Management Information Base : see Section 2.4 on Page 12.

NAT Network Address Translation is a technique in which the source and/or destination addresses of IP packets are changed to reflect the IP address of the inside or outside network as they pass through a router or firewall.

OID Object Identifier : see Section 2.3 on Page 10.

PDU A Protocol Data Unit is a unit of data that is specified in a protocol and that consists of protocol-control information and user data.

RFC A Request for Comments is an informational document about Internet standards and protocols.

SNMP Simple Network Management Protocol : see Chapter 2.

TCP The Transport Control Protocol is one of the core protocol of the transport layer. This protocol permits, among other things, different programs on computers connected to a network to create a reliable connection.

TFTP Trivial File Transfer Protocol : Does the same as FTP, but has no authentication and no possibility to browse folders on a target server.

UDP The User Datagram Protocol is, like TCP, a protocol of the transport layer, but does not create a reliable connection.

VeriNeC Verified Network Configuration [13] : see Chapter 3.

XML eXtensible Markup Language [21] : see Section 4.2 on Page 28.

XSLT eXtensible Stylesheet Language Transformations [23] is a language for transforming XML documents into other documents like text, html, PDF or XML.

Bibliography

[1] Douglas Mauro and Kevin Schmidt. *Essential SNMP*, First Edition. O'Reilly, July 2001.

[2] Todd Lammle. *CCNA: Cisco Certified Network Associate (Study Guide)*, Fourth Edition. Sybex, 2004.

[3] Todd Lammle, Sean Odom and Kevin Wallace. *CCNP: Routing (Study Guide).* Sybex, 2001.

[4] Cisco System, Inc. . *Cisco 826, 827, 828, 831, 836, and 837 and Cisco SOHO 76, 77, 78, 91, 96, and 97 Routers Software Configuration Guide* . 2003.

[5] Dominik Jungo, David Buchmann and Ulrich Ultes-Nitsche. *The role of simulation in a network configuration engineering approach.*

[6] David Buchmann. *Verinec Translation Module*. Working Paper.

[7] David Flanagan. *Java in a nutshell*, Fourth Edition. O'Reilly, March 2002.

[8] Andrew S. Tanenbaum. *Computer Networks*, Fourth Edition. Prentice Hall, 2003.

[9] Elliotte Rusty Harold and W. Scott Means. *XML in a nutshell*, Second Edition. O'Reilly, June 2002.

[10] Brett McLaughlin. *Java & XML*, Second Edition. O'Reilly, August 2001.

[11] Doug Tidwell. *XSLT*, First Edition. O'Reilly, August 2001.

[12] Eric van der Vlist. *XML Schema*, First Edition. O'Reilly, June 2002.

[13] Verified Network Configuration project,
`http://diuf.unifr.ch/tns/projects/verinec`

[14] Netopeer,
`http://www.liberouter.org/netopeer`

[15] SNMP implementation for Java,
`http://www.snmp4j.org`

[16] Net-SNMP,
`http://www.net-snmp.org`

[17] SNMPLink,
`http://www.snmplink.org`

[18] mibDepot,
`http://www.mibdepot.com`

[19] Cisco Systems,
`http://www.cisco.com`

[20] Wikipedia.org,
`http://wikipedia.org`

[21] Definition of XML,
`http://www.w3.org/XML`

[22] Definition of XML Schema,
`http://www.w3.org/XML/Schema`

[23] Definition of XSLT,
`http://www.w3.org/TR/xslt`

[24] Eclipse, IDE platform for Java,
`http://www.eclipse.org`

[25] NetworkComputing.com,
`http://www.networkcomputing.com`

[26] Java Developers Almanac,
`http://javaalmanac.com`

[27] Java homepage,
`http://java.sun.com`