Implementing NAT support into the XORP project.

M.Sc. Thesis

Kristen Nielsen
kristen@diku.dk, krn@krn.dk

Marts 29, 2008
Final Version 1.1

Department of Computer Science
University of Copenhagen
Denmark

Contents

List of Figures

Acknowledgments

During the project time (January 2006 to Marts 2007), I have got great support from the people who shared the S.226 office at Diku with me, with whom I shared the lunch and coffee breaks and who was there to discuss ideas of the day, artifacts of the C++ languages and other topics on my mind during the time of the project but most of all for just being there. The persons I specially would like to thank in this respect are: Pelle Kristian Lauritsen, Rasmus Melgaard, Jens Blåbjerg, Martin Bjelke, Steffen Ramsøe, Cail Christoffer Buhl Borrell, Steffen (Zool) Larsen, Jesper Hansson, Berit Løfstedt, Anja Lee Pedersen and Tor Bech Petterson and the many people who only was there a shorter period during the project period who will remain unnamed here.

I also want to thank my Project advisor at DIKU Jørgen Sværke Hansen, for his proper and valuable feedback of the draft version of this report and for being there when I needed him during the project.

Kristen Nielsen
Marts 10, 2007.

During 2007 the present project was administratively upgraded to be my Master Thesis work mainly due to the fact that the time and effort I had spent on the project qualified for this. During this time Professor Eric Jul was my adviser as Jørgen Sværke Hansen meanwhile had left DIKU. I would also like to thank Professor Eric Jul for his time and effort during the second round of the project and for helping with all kinds of project related guidance and administrative tasks regarding the upgrade of the project to the Master Thesis project.

The changes to the report from the first version (Marts 2007) are only some extra rounds of additional proofreading and adding the Master Thesis front page and DIKU logo.

Finally being close to the end of almost 12 years of being a spare time student at DIKU, a special thank from deep inside my heart goes to the Clerk of Study Administration Ms. Ruth E. Schlüter for always being there as the big time smiling person in her office and for an almost endlessly number of comments and inspiring views of daily events, for having time to answer almost any kind of questions I have had and in general for being the person she are. Since the first week as a student back in September 1996 and until now I have received much qualified help from Ruth and we have had many great times and laughs together, you were always there when I needed you – Thank you Ruth.

Kristen Nielsen
January 20, 2008
Department of Computer Science - DIKU
University of Copenhagen

Version list.

Final version Marts 10, 2007.
Final Version January 20, 2008. The Master Thesis version handed in at Diku.
Final version 1.1, Marts 29, 2008. Minor typos corrected.

Chapter 1

Introduction and Conclusion

Why this project?

We have chosen this project as we would like to evaluate the eXtensible Open Router Platform (XORP) as the platform for our future master thesis project. Making a smaller project with the XORP platform before using it in our master thesis project will give us a better understanding of the XORP system, how it is put together and how it works internally. We have chosen the Network Address Translation (NAT) project because it seemed as a small project, and because NAT functionality is not yet implemented in the XORP project.

By working with an already established source code base, we believed that we would not risk falling into the ”we only use the C++ code technologies we already know about” trap. So by choosing an established project with a substantial code base we expect to learn new ways of C++ coding and to get inspiration from how other programmers are using the language.

Last but not least we believe that working with a real project is of great value to the enthusiasm we put into the work, as it may be used by other people for real things and not only be “our” project.

A few initial words about XORP and our NAT project

XORP out of the box, has no support for NAT, which in our opinion is need-to-have functionality for an Internet router to be usable for most Internet users. The XORP project has a vision of having a rich feature set enabling it to function as a backbone router, at least somewhere at the edge of core backbone networks. This means that all functionality of main Internet backbone routers must be supported by XORP routers too.

Routers located at core networks are not using NAT a whole lot, but routers located at the edge of core network or at customers premises does often use NAT. Having NAT the XORP router will be much more usable for low end or home users, which will help acquiring a larger user base and give XORP a wider deployment.

If NAT is going to be used with XORP today (before we implement NAT support), NAT has to be configured at the supporting operating system and then the XORP configuration file must be configured in such a way that it works with the pre-configured NAT of the host operating system.

During this project we will add functionality to the XORP configuration language and write a NAT module for XORP that handles the NAT part of XORP. As a first try we will use the native NAT on FreeBSD, called “natd”, later we will plan adding support enabling the same XORP NAT configuration to be implemented (executed) by the Click project [18] IP forwarding subsystem. This will give us more sophisticated NAT functionality which we will plan with during the project.

Conclusion

A first proposal for the NAT module interface and the configuration syntax has been discussed with the XORP project, and the outcome of this led us to choose the "protocol" and "sub protocol" terms, as they requested a more generic framework that supported all protocols that can be carried by an IP packet and not only the TCP/UDP ports. This combined with the large number of parameters needed to configure the 3 types of NAT (static, dynamic and load-sharing) we are planning to have support for, directed us towards the path of putting effort into designing new syntaxes for XORP to ease the specification of IP addresses, IP subnets and IP ranges as well as protocol and port numbers (which we call sub-protocols) and finally adding the ability to add interface and virtual interfaces to the configuration language. All this reduces the number of parameters for a NAT interface from 7 to one and highly reduces the complexity of our configuration language and gives better readability of XORP NAT configurations.

The only thing we did not add to the defined syntax is the possibility to define a realm which we probably will/should do at a later time as it appears to be a relevant extension.

We have used a large amount of time to get acquainted to the XORP code and how things are working internally in XORP. We also used quite some time on trying to get the Click code to work at the FreeBSD 6 release kernel, but had to go back using FreeBSD release 4.9 to get Click support in kernel mode. We have put great effort into creating a generic framework for configuring NAT with support for all the many flavors NAT comes in as we learned during the analysis of existing NAT implementations. We decided on having a generic configuration syntax and then knowing that if a given configuration can not be "produced" by a given NAT module (in the IP path) we will issue an error message to inform about this.

We have implemented all described functionality, the full syntax as shown in section 5.8 at page 51 except the configuration generator module and the code to implement the `fea {}` clause of the configuration language. We still need some more testing to be done before releasing the code to the XORP project. The new syntaxes for protocol and sub protocols we proposed in section 5.6 and 5.7 has not been implemented during the project.

It has been an interesting project but we clearly underestimated the effort needed to get acquainted with the XORP code base and the effort we had to put into designing a general configuration language for the module. At the same time we learned that our C++ knowledge was not as updated as we believed it was, so we had to get newer versions of the C++ books to understand the new syntaxes used in the C++ of today.

The project has been really interesting to do and we hope that the XORP project will use some of what we created and import it into their code base.

Kristen Nielsen
Diku / Computer Science Department, University of Copenhagen
kristen@diku.dk, krn@krn.dk
10. Marts 2007.

Chapter 2

An introduction to the XORP project

This chapter introduces the eXtensible Open Router Platform (XORP) to the reader unfamiliar with the XORP project and the overall design of XORP. We provide a short introduction to the XORP project goals and to each of the modules which is a part of XORP. Finally we provide a short description of how to implement or add a new module into the XORP project.

This chapter is inspired by the XORP document "XORP Design Overview" [1] and the "XORP Software status page" [2].

2.1 XORP overview

The XORP homepage[1] says it like this:

> "XORP is the eXtensible Open Router Platform. Our goal is to develop an open source software router platform that is stable and fully featured enough for production use, and flexible and extensible enough to enable network research. Currently XORP implements routing protocols for IPv4 and IPv6 and a unified means to configure them. In future, we would also like to support custom hardware and software forwarding architectures."

XORP has been around since January 2003, where version 0.1 was released, the XORP project has on a regular basis released versions with enhanced functionality. Today (November 2006), we have release 1.3 of XORP. XORP have a good overall design documentation which will be referenced throughout this report, but to introduce the reader to the overall design of XORP we have included a (not so) short introduction to the XORP design and modules here.

XORP is running on top of a host operating system (Host OS), which natively is FreeBSD but Linux (more distributions), Mac OS X and even MS Windows 2003 Server are current possible host operating systems. Some XORP features might not yet be implemented at some of the host OS'es.

Generally XORP has modularity as the overall design concept. Every module of the router is a separate Unix process at the host operating system. To facilitate future hardware with more processors or dedicated packet forwarding subsystems all communication between XORP modules is done via a unified communication system that facilitates modules can be executed at different hardware which are connected with some kind of communication path e.g. an IP network, or a high-speed internal backplane bus between the individual

[1] http://www.xorp.org

processors. In XORP communication between modules is done with eXternal Resource Locators (XRL).

XRLs looks much like URLs known from the World Wide Web. Figure 2.1 shows a example of an XRL and explains the various parts. The Protocol Family, the Interface specification and the arguments.

```
finder://fea/fti/0.1/add_route?net:ipv4net=10.0.0.1/8&gateway:ipv4=192.150.187.1
+-----   +-- +-- +-- +-------- +-----------------------------------------------
|        |   |   |   |         |                      ======= ==== =============
|        |   |   |   Method    Arguments                ^       ^          ^
|        |   |   |                              Parameter name -|       |          |
|        |   |   Interface version                                     |          |
|        |   |                                  Parameter type --------|          |
|        |   Interface Name                                                      |
|        |                                      Parameter value ------------------|
|        Protocol Parameters
|
Protocol Family
```

Figure 2.1: Example of an XORP external Resource Locater (XRL). The figure define the various parts of an XRL. Arguments starts with the question mark "?" Character and are separated with the ampersand Character "&". The base example is from the XORP Inter-Process Communication Library Overview document [5].

XORP is deliberately not a multi-threaded architecture which reduces the design requirements and complexity of the individual XORP modules and eliminates the limitations that a multi-threaded architecture adds to the programming rules.

2.2 XORP modules

XORP consists of a number of modules. Every XORP module is implemented as a separate operating system (Unix) process. Throughout this chapter we use the term module but process would have been an equal correct term to use here. The modules are divided into 3 main groups: Management modules, Unicast modules, Multicast modules and the few modules that don't fit into these classes are grouped under the "Other modules" group. The individual XORP modules/processes are depicted in figure 2.2 and described in the remaining parts of this section:

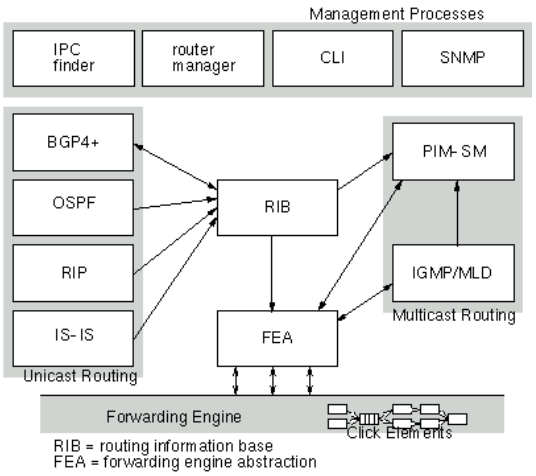


Figure 2.2: XORP processes overview. The figure is from the XORP Design Overview document [1].

Management modules

Inter-Process Communication (IPC) Finder

The IPC finder is needed by the XRL communication method used among all XORP components. Each of the XORP components registers with the IPC finder. The finder assists the XRL communications by knowing the location of each XRL target, therefore a XORP process does not need to know explicitly the location of all other process, or how to communicate with them. The router manager process *(rtrmgr)* incorporates a finder, so a separate finder process is only needed if the *rtrmgr* is not being used such as during testing. For more information about the IPC finder and XRLs see these XORP documents: "XORP Inter-Process Communication Library" [5] and "XORP XRL Interfaces: Specification and tools" [12].

XORP Router Manager *(rtrmgr)*

The *rtrmgr* is the XORP process responsible for starting all components of the router, to configure each of them, and to monitor and restart any failing process. It also provides the interface for the Command Line Interface (CLI) to change the router configuration. For more information about the *rtrmgr* see "XORP Router Manager Process" [10].

Command Line Interface (CLI) *xorpsh*

The CLI can be used by a user to access the router, view its internal state, or to configure it on-the-fly. Its functionality is closely related to the *rtrmgr*. However, because the robustness of the *rtrmgr* itself is extremely important, all functionality that can be run as a separate CLI process are separated from the *rtrmgr*. The process implementing this CLI functionality is called *xorpsh*. For more information about the CLI and the *xorpsh* process see "XORP User Manual" [13].

Simple Network Management Process (SNMP) module

This is the SNMP management process of XORP. It is used for SNMP access to the router. For Example, it can be used to translate SNMP requests into XRL requests. Internally, SNMP will communicate with the other processes using XRLs. For more information about the XORP SNMP module see the document "XORP SNMP Agent" [14].

Routing Policy module (Policy) module

This is the XORP routing policies coordination process[2] The policy module basically works by checking the routes or other data that are policed by the module ensuring that the global policies are respected. As an example: if import of routes from OSPF into BGP are policed by the module, each of the involved modules can ask if a given route is blocked or allowed by the current policy. The policy module interacts with all routing protocols and the Routing Information Base (RIB) and instructs them how to handle the routes flowing to or from the system, export rules from one protocol to another, modify or remove routes as they flow through the system, etc. Currently only unicast policies are supported by the policy module. Policy is a newly added feature of XORP and the number of modules that implements policy functions is growing as this is currently being implemented. For more information about the XORP policy module see the paper: "Decoupling Policy from Protocols: Implementation Issues in Extensible IP Router Software" [16].

[2]The Routing Policies module is not shown on the figure 2.2.

Unicast routing modules

Border Gateway Protocol version 4+ (BGP4+) module

This is the BGP routing daemon. It implements IPv4 and IPv6 unicast routing in a single process, as well as Multicast BGP4 (MBGP4) for both IPv4 and IPv6 multicast RIB for multicast routing purpose. For more information about the BGP see "XORP BGP Routing Daemon" [3] and rfc-4271 [35].

Open Shortest Path First (OSPF) module

This is the OSPF routing daemon. There are separate IPv4 and IPv6 daemons, because unlike BGP there is no real need to tie them together. For more information about the OSPF module see the XORP website and the source code and rfc-2328 [29]. No separate XORP document about the OSPF implementation has yet been released from the XORP project.

Routing Information Protocol (RIP) module

This is the RIP routing daemon. Similarly to OSPF, the IPv4 and IPv6 daemons are separate. For more information about RIP see "RFC-2453. RIP Version 2." rfc-2453 [30].

Intermediate System - Intermediate System (IS-IS) module

This is the IS-IS routing daemon. The is-is routing protocol derives from the OSI network specification. The XORP IS-IS project is not yet published.

Multicast routing modules

The overall design of the multicast routing system is described in the XORP design document "XORP Multicast Routing Design Architecture" [8].

Protocol Independent Multicast – Sparse mode (PIM-SM) module

This is the PIM-SM module. For more information see the "PIM-SM Routing Daemon" document [9].

Internet Group Management Protocol / Multicast Listener Discovery (IGMP/MLD) module

This is the MLD/IGMP handler. It implements the router-side part of MLD and IGMP protocol. Its main purpose is to discover local multicast members and propagate this information to multicast routing daemons such as PIM-SM. The IGMP (IPv4) and MLD (IPv6) daemons are separate implemented modules. For more information about the XORP MLD/IGMP implementation see "XORP MLD/IGMP Daemon" [6].

Other modules

Forwarding Entity Abstraction (FEA) module

The Forwarding Entity Abstraction (FEA) is a common interface for XORP towards the packet forwarding hardware. The FEA controls all network interface cards at the hosting OS (e.g. issuing proper `ifconfig` commands etc.), installs multicast support, and is also responsible for the communication with the Click modular router system if installed.

By definition XORP network interfaces has a two level hierarchy. The real (physical) interfaces are called `interfaces` and virtual interfaces are called `vif`. All IP setup is done at the `vif` level. If an physical interface like an Ethernet NIC is capable of handling virtual LANs (VLAN) these are defined as a sub interface of the physical interface as a `vif` for each VLAN. If no virtual layer is existing at an real interface, a vif must still be configured to define the IP-interface for this interface card. The names for interfaces are usually the names used by the hosting OS for a given interface. `Vif` names are usually the same with an added decimal number to identify the VLAN.

The multicast-related functionalities are logically separated from the unicast-based functionalities in the Multicast Forwarding Engine Abstraction (MFEA), though the MFEA is part of the FEA process.

The FEA is described in more detail in the XORP document: "XORP Forwarding Engine Abstraction" [4], and the MFEA is described in the XORP document "XORP Multicast Forwarding Abstraction" [7].

Routing information Base (RIB) module

The RIB holds a user-space copy of the entire routing/forwarding table, complete with information about where each route originated from (e.g. which protocol, and when). For more information about the RIB see " XORP Routing Information Base (RIB) Process" [11].

Forwarding Engine (FE) subsystem

The forwarding engine is the underlaying IP subsystem that handles the actual IP packet stream that passes through the XORP host. It is usually the default IP subsystem of the XORP host, but it can also be a Click modular router module that handles the packets being routed. Theoretically it can also be custom made IP subsystem, e.g. implemented at a multi port Network Interface Card (NIC) which is capable of routing IP packets between its local ports - and this way off-loading the IP subsystem of the XORP host.

When the native NAT of FreeBSD (`natd`) is used in our project we will have to instruct the IP subsystem to divert IP packets to the NAT module that lives as a user land process, communicating with the IP subsystem via a socket. Commands for the IP divert, a FreeBSD kernel module, can specify an ACL that selects the IP ranges that is diverted through the `natd` module. We will as a default have all packets diverted through the `natd` module.

Click modular router

The Click modular router project has been written by Eddie Kohler and is described thoroughly in his PhD thesis [19]. The Click project homepage [18] is carrying all documentation about the project.

The Click modular router is a router which has modules as the basic building blocks. Click modules shares a common interface structure, which enables modules to be connected together as described in the configuration file - in coarse the principle is similar to a number of UNIX shell commands piped together. The ability to combine small function modules into a system that e.g. works as an IP packet forwarding system, a NAT device, a Switch or an IP router. It has the possibility to work both at the Ethernet layer (layer 2) and IP packets layer (layer 3) and higher layers such as UDP or TCP. A library of well written and tested modules is distributed with Click easing the startup learning curve. Click has a configuration language to describe the desired configuration. The language describes which modules are connected together and in this way describes the desired module setup or system function.

On a few platforms Click are implemented as a loadable kernel module and on the rest of the supported platforms, click is only supported as a user land module that communicate

with the underlaying IP subsystem with methods like IP divert. Both implementations has the same set of features, understands the same configuration files and loads the same Click modules - only the speed differs between kernel and user land variations. XORP already has support for specifying Click configurations in the FEA module.

Our intent is to use the click IPRewriter [20] module as the advanced NAT function device supported by this project. Click has a number of modules that is designed to perform packet rewriting and NAT/NAPT (explained later) translation of IP packets. We might need to use a few other modules than the IPRewriter module to implement full NAT functionality.

2.3 XORP system overview

The XORP system has a overall manager module, called The XORP route manager *(rtrmgr)*, its main task is to read the XORP configuration file, interpret it and communicate the contents to the rest of the modules in the XORP router. To produce the setup described in the XORP configuration file, the router manager builds an "in-memory" configuration tree where each configuration clause is stored in its own node. The configuration tree holds the running configuration of the router. Users can interactively via the XORP shell *(xorpsh)* change the content and hence the running configuration of the XORP router, e.g. if an IP address of an interface is changed in the in-memory configuration tree, the XORP *rtrmgr* communicates this to the FEA which issues a proper command to the underlaying hardware to change the IP address of the interface.

Templates for the *xorpsh* and *rtrmgr*

The *rtrmgr* and the *xorpsh* uses templates that describes functionality for each module included in the XORP project.

Templates defines the structure of the configuration tree for each node and the names and types of the parameters. Templates contains a header and a number of sections.

The header defines which other modules the current template depends on. This is used by the *rtrmgr* to startup the needed modules only.

One section contains the information about what actions to issue to the various XORP modules to implement the functionality expressed in a given in-memory configuration node. Actions can typically be: establishing, changing or deletion of functionality contained in a given node and what XRL calls to issue for each node to achieve the desired action. Another section of the template file contains help texts (short and long versions) for the interactive user using the *xorpsh* CLI interface program.

Templates control the syntax and parameter types of each node in the in-memory configuration tree, allowed value ranges for each parameter and what actions (XRL commands) to be executed when configuration parameters are added, deleted or changed. The template also describes where in the configuration tree the parameters for each XRL call issued to a module are stored (or fetched).

XORP Shell *(xorpsh)*

The user interface for XORP is the xorp shell which is implemented in the program called *xorpsh*. The XORP shell connects to the *rtrmgr* to get access to the in-memory configuration tree to access the current configuration of the XORP system and also uses the templates that describes each modules. The XORP shell implements the CLI through which the users controls and configure the XORP system.

When a node or a parameter is crated, edited or deleted the XORP shell notifies the *rtrmgr* which then perform the changes in the in-memory configuration tree and calls the relevant actions according to the nodes template file to implement these.

2.4 A short introduction to write a XORP module

To get an idea of what steps are involved in writing and adding a new module to the XORP project, we will try to give a brief overview of this in this section. The tasks involved in implementing a new XORP module are:

- Write a configuration syntax for the module functionality.
- Design an module interface and write a XRL interface file to the module. (`.xif and .tgt files`)
- Write a *rtrmgr* template for the module. (`.tp file`)
- Write the XORP module code that implements the desired functionality when configured via the XRL interface.
- Eventually implement the needed (extra) functionality in the FEA module and other modules that directly calls the module. (This part is not described further in this project.)
- Eventually write helper tools for the *rtrmgr*. These are most often used for output listings of e.g. interface setups etc. (This part is not described further in this project.)

Writing a configuration syntax

A XORP configuration syntax determines the structure of the syntax and configurable elements of the XORP module. XORP configuration syntax is modelled over a tree structure. At each branch in the tree either one single "unnamed" node or a number of named nodes can exist. At the same time each branch (node) can have can have a number of named leaves which we call parameters in this project. Each leave contains one typed data variable. A small example from the configuration syntax for the NAT module is given in figure 2.3. The example defines a `status-native-nat:` parameter that returns a boolean parameter named "ok" in the `fea{}` clause. The `status-native-nat:` parameter returns the status of the natd subsystem as a boolean parameter. The example also defines a `nat id:txt {}` clause with the name stored in the text parameter "id:". The `nat {}` clause contains a new named node `native-nat id:txt {}` with the name stored in the text parameter "id:" and which has 2 parameters: a boolean parameter named `disable:` and a text parameter named `natd-command:`. The tree structure and the node and parameter names described in this syntax, matches the structure that is used in the in-memory configuration tree, in the CLI and in the configuration file.

```
fea {
    status-native-nat: <ok:bool>

    nat <id:txt> {
        native-nat <id:txt> {
            disable: <disabled:bool>
            nat-command: <nat-command:txt>
        }
    }
}
```

Figure 2.3: Example of the XORP configuration syntax. The language syntax used in the description is not a formal specification of XORP.

Designing a XORP module XRL interface

Writing a new XORP module requires the design of an XRL interface that fits the desired functions of the module. This task is similar to normal software design principles. You can freely choose to have many small functions or fewer larger functions. We have the fundamental idea designing our XRL that each XRL calls should contain a unit of knowledge, which is somehow self-explaining in the context it is described in. E.g. a XRL call to setup a NAT translation rule, holding/releasing the configuration generation mechanism. In some modules this resolves to a large number of XRL interfaces and in other it resolves to fewer interfaces.

During the next pages we will give an example of how our XRL interface design is done by explaining a small part of the *rtrmgr* template file for the `fea natd` interface. We describe the configuration syntax used in figure 2.3 at page 9 and what each part is doing. This (we hope) gives an overall idea of the function (or role) of the template file.

Writing an XRL interface

XORP is a modular structured and template based system where each part of XORP is a self contained Unix process that communicates with the remaining parts of the XORP system via XRL calls. The XRL interface of each module is defined by an interface description file with the extension `.xif` The list of other module names that a given XORP module is communicating with is defined in a target description file `.tgt`. From these two files XORP generates module interfaces for the XRL communication to and from the module and the matching stub code to include into the calling modules. Code for marshalling and unmarshalling of the parameters sent via XRL calls between XORP modules are also generated from these 2 files. The `.tgt` file is not further described here.

The XORP module interface is modelled from the configuration syntax and the design of the new module. The basic idea is that each XRL call is creating a self-contained unit of configuration knowledge, that either is defined or not defined as a whole. That is why we chose to combine the `nat {}` and `native-nat {}` nodes into one interface description file (`.xif`). A short overview of the XRL interface description language used in the `.xif` file is given in figure 2.4 below. An example of an `.xif` file is given in figure 2.5 below. Each interface description is contained on a single line. If the definition is too long a `<backslash><new-line>` character sequence is used as a non terminating `<new-line>` character (which is an accepted standard method used in ASCII files). The whole language is defined in the XORP technical document "XORP XRL interfaces: Specification and Tools." [12].

```
interface          ::= <interface-name>/<interface-version> { <xrl-method-list> }
interface-name     ::= <identifier>
interface-version  ::= <digits>.<digits>
xrl-method-list    ::= <xrl-method> [<xrl-method>...]
xrl-method         ::= <function-name> [?<argument> [&<argument>...]]
                       [ -> <return-parameter> [&<return-parameter>...]]<new-line>
function-name      ::= <identifier>
variable-name      ::= <identifier>
return-parameter   ::= <variable-name>:<type>
identifier         ::= <letter>[[<letter>|<digit>]...]
argument           ::= <variable-name>:<type>
type               ::= u32|u32range|i32|txt|bool|toggle|ipv4|ipv4net|ipv4range
                         |ipv6|ipv6net|ipv6range|macadder|com32
```

Figure 2.4: A brief overview of the XRL interface description language used in the `.xif` files.

```
interface natd/0.1 {

    status_natd ->  ok:bool
    create_native_nat ? id:txt & disabled:bool & natd_command:txt
    delete_native_nat ? id:txt & disabled:bool & natd_command:txt
}
```

Figure 2.5: Example of the XORP `.xif` file syntax. The example shows the `status_natd` call that has a boolean `ok` parameter which is returned to the calling function and the `create_native_nat` and `delete_native_nat` calls which each takes 3 arguments from the calling module.

Writing a template for the module

We provide a small XORP template example that shows the template matching the above examples. Although it might seem a bit complicated the example is a good way to gain some basic understanding of what functionality and mechanisms the XORP template files provides.

Figure 2.6 shows the template matching the examples from figure 2.5. The example defines the in-memory (and configuration file) tree-node layout which is explained in details below:

Line 1–10: Defines the node-tree syntax and the data type of each node. Line 12–49: Defines the properties, help texts and actions to execute (by the *rtrmgr*) when each node is changed.

Line 13: Informs the *rtrmgr* that the current file provides the functionality in the XORP system called "natd". Line 14: Declares the functionality defined in the current file depends on the functionality in the module named "fea". Line 15: Tells the *rtrmgr* where to find the executable for this xorp module at runtime. Line 16: Sets the default name for the `targetname` node in this module. The `targetname` node contains the name that the module uses when identifying itself to the *rtrmgr* – this is a XORP convention. Line 17: Instructs the *rtrmgr* that the `targetname` parameter has to be set before the module is started up. Line 19–20: Defines the functions for the `targetname` in-memory node. The `%set:;` command indicates that `%set:` is the (only) operation allowed at the `targetname` node. There is no xrl command after the ":", which means the value supplied with the command or configuration will be set as the node value. If the `%set:` was not specified then no `set` command is allowed at the node. If a xrl command was specified this command is issued when the set operation is issued from the *xorpsh* CLI program (or when loading a configuration file). Line 30: Sets a short (one-line) help text displayed to the interactive user by the help command of the *xorpsh* CLI program.

Line 31: specifies the command that the *rtrmgr* executes when a `native-nat` node (line 29–46) is created. The `$(...)` clauses refers to parameter substitution from the in-memory configuration tree. The `@` is the `id` parameter of the current node (here it represents the `native-nat` node name (a text parameter) from line 29). If the `targetname` node contains "natd" the evaluation of the macro will give:
"fea.natd/natd/0.1/create_native_nat?id=natid&disable=false&natd-command=<natd-command>".
The <natd-command> is the actual "natd-command" that is the argument to the "natd-command" parameter. The `<new-line inserted>` (line 32 and 35) are editorial inserted line breaks which does not exist in the real file. Line 34: Is the xrl command executed when a `native-nat` node (line 29–46) is deleted. Line 40: Is the command executed when a get operation is performed on the node. This statement is empty which implicates that a `%get:` operation will just return the value of the node. Besides the listed operations a wider range of commands and parameters exists in the template vocabulary. The full set is described in the XORP technical document "XORP Route Manager Process *(rtrmgr)*" [10].

```
01: fea {
02:     status-native-nat: bool;
03:
04:     nat {
05:         native-nat @: txt {
06:             disable: toggle = false;
07:             natd-command: txt;
08:         }
09:     }
10: }
11:
12: fea {
13:    %modinfo: provides natd;
14:    %modinfo: depends fea;
15:    %modinfo: path "nat/xorp_nat";
16:    %modinfo: default_targetname "natd"
17:    %mandatory: $(@.targetname)
18:
19:    targetname {
20:        %set:;
21:    }
22:
23:    status-native-nat {
24:        %help: short "returns true if natd subsystem is available on host.":
25:        %get: xrl "$(fea.targetname)/natd/0.1/status_natd->ok:bool;
26:    }
27:
28:    nat {
29:        native-nat @ {
30:            %help: short "specifies a native-nat configuration"
31:            %create: xrl "$(fea.targetname)/natd/0.1/create_native_nat?
32:             <new-line inserted> id=$(@)&disabled=(@.disabled)&nat-command=$(@.nat-command)";
33:
34:            %delete: xrl "$(fea.targetname)/natd/0.1/delete_native_nat?
35:         <new-line inserted> id=$(@)&disabled=(@.disabled)&nat-command=$(@.nat-command)";
36:
37:            disabled {
38:                %help: short "disables the native-nat.";
39:                %set:;
40:                %get:;
41:            }
42:            nat-command {
43:                %help: short "send a command to the native-nat.";
44:                %set:;
45:            }
46:        }
47:    }
48:}
```

Figure 2.6: Example of a XORP template file `.tp`. The "<new-line inserted>" markers marks line breaks that is inserted to format very long lines to the page width, and is not a part of the real file.

Writing the XORP module code

What properties are required for XORP module code? The XORP system puts a number of requirements onto modules to fit into the system. Here is a list of the most important requirements with a short description of each:

XRL interface: The module code must fit together with the C++ classes defined for the XRL interface. The code for the XRL interface is automatically generated from the interface description `.xif` and the target description `.tgt` files by the XORP build process. The code also define virtual function definitions for each xrl interface defined in the `.xif` file, that has to be implemented by the module programmer to perform the desired operation at the module. The overall concept is that when some XORP module generates an XRL call to our module the function pointed out by the XRL call in our module is called (executed) with the parameters received. When the module has done its work the function returns with results through parameters according to the interface description. As XORP is a non multi-threaded architecture the module is not allowed to block. This puts a requirement on the code that if it can not finish its task immediately it has to return the XRL call immediately and use a callback method to supply the results later. When return parameters are ready they are returned to the (originally) calling instance via a registered call-back function (a sort of a XRL pointer). Call back is an integral part of the XRL functionality and an automatic generated feature of the XRL system. We do not further describe XRL callbacks in this report but refer to the XORP document "An Introduction to Writing a XORP Process" [15].

XRL error codes: XRL calls has a few status codes that is returned with each call. The most often used code is the:

`OKAY` code which is the everything-is-normal return code which transfers the call parameters to the other end.

All other return codes are error indicating codes indicating that the parameters is not transferred to the other end. A few examples of theese are: `BAD_ARGS` indicating that arguments are not valid, `COMMAND_FAILED` indicating the XRL command failed, `NO_FINDER` indicating the finder is not found, `RESOLVE_FAILED` indicating no receiver address is found and `NO_SUCH_METHOD` indicating that the method (function) stated in the xrl is not found.

All error codes are indicating a serious error which signals that the XRL did not execute normally at the other end. The full list of XRL error codes is found in the `$XORP/libxorp/xrl_error.hh` file where `$XORP` represents the base directory of the XORP source code.

Process model: XORP has its own process model which is depicted in figure 2.7 that describes which states a process can be in, and what transitions it can perform between each of these. The process model is described in the source file `$XORP/libxorp/status_codes.h`. The following process states exists:

`PROC_NULL`: Process is not registered with the finder. It may or may not be running.

`PROC_STARTUP`: The process is registered with finder, but is waiting on some other processes before it is ready to be configured.

`PROC_NOT_READY`: For any reason, the process is not ready for processes that depend on this process to be configured or reconfigured. A common reason is that is this process just received a configuration change, and is still in the process of making the configuration change active.

`PROC_READY`: Process is running normally. Processes that depend on the state of this process can be configured or reconfigured.

PROC_SHUTDOWN: Process has received a shutdown request is shutting down cleanly. Normally the process till terminate by itself after being in this state.

PROC_FAILED: Process has suffered a fatal error, and is in the process of cleaning up the mess. Normally the process will terminate by itself after being in this state.

PROC_DONE: The process has completed operation, but is still capable of responding to XRLs.

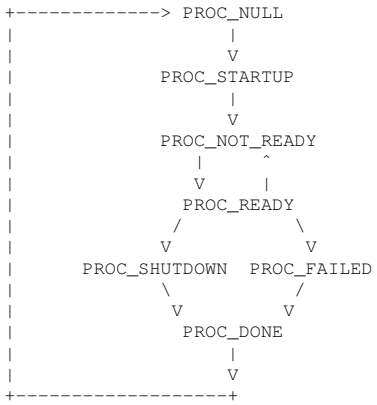


Figure 2.7: The XORP Process Model and the names of each process state. The figure is from the source file $XORP/libxorp/status_codes.h

During the recent pages we have provided a short overview of the insights of writing a XORP module. We have chosen to describe the parts needed to understand the discussion and the description of the implementation of the XORP NAT module without going into a too detailed level of XORP technicalities. We have tried to reference relevant documentation for topics which are further described in the XORP documentation.

Chapter 3

A Model of Network Address Translation (NAT)

Introduction

The first part introduces and defines a number of terms used throughout this report. The second part introduces Network Address Translation (NAT) terms and definitions and a number of NAT flavors. The third part introduces Network Address Port Translation (NAPT) and explains the details with a number of examples. The fourth part describes load-sharing NAT (LS-NAT) with detailed examples. Finally we provide a short recap of the entire chapter.

3.1 Definition of terminology

In this section we will introduce and formalize the terms and expressions that we use throughout this chapter.

IP address

IP-address is meaning a single IPv4 or IPv6 address. Mostly we only consider IPv4 addresses as NAT is not really applied to IPv6 network technology.

IP subnet mask / subnet mask

The IP subnet mask is a bit pattern that defines what parts of an IP-address are used for the network address and what parts are used for the host address. Subnet masks consists of 24 bits (IP version 4) with the property of a row of 1-bits followed by a number of 0-bits. The 1-bits in the subnet mask tells that the corresponding bit in the host IP-address belongs to the network-address, and the 0-bits tells that the corresponding host IP-address bits belongs to the host IP-address of the IP-subnet.

The subnet mask only has local significance - which means that only the hosts and routers connected to a specific IP-subnet needs to know about the subnet mask. All hosts on a IP-subnet need to share the same subnet mask to have the same understanding of what IP-subnet they belong to (as a change in the subnet mask changes the number of bits in the IP-subnet address).

Subnet masks for IPv4 can be written in 2 different ways: As 4 groups of 3 digit decimal numbers separated with dots e.g. `255.255.255.0` or as a 2 digit decimal number after a slash stating the number of 1-bits in the subnet mask e.g. `/24`. The latter version is often

printed together with an IP-address like this: `10.200.18.1/24`, meaning the IP-address `10.200.18.1` with a 24 bits subnet mask `(255.255.255.0)`.

IP-subnet / IP-network

IP-subnet and IP-network is both used with the same meaning throughout this report. A IP-subnet is a range of IP-addresses starting at the IP-network address and ending with the IP-broadcast address for the IP-subnet.

An IP-subnet is specified with an IP-address and a subnet mask. The number of IP-addresses in the IP-subnet is calculated as the power of 2 to the number of 0-bits in the subnet mask. (e.g. subnet mask `255.255.255.0` has 8 0-bits hence 256 IP-addresses in the IP-subnet). Two of these are reserved, the first IP-address for the IP-subnet address and the last (highest) IP-address for the IP-broadcast address - these can by definition not be assigned to any hosts at the IP-subnet.

IP-subnet address

Is by definition the first (lowest) IP-address of an IP-network and is not to be assigned to any host. The IP-subnet address has by definition all its host IP-address-bits set to "0".

IP-broadcast address

The IP-broadcast address is the last IP-address in an IP-subnet. The Broadcast address is used for sending packets to all IP hosts at the IP-subnet. By definition no host can have the address of the IP-broadcast address, and the IP-broadcast address has all its the host IP-address-bits set to "1".

IP-range

An IP-range is a range of IP numbers defined by a low and a high IP-address. The IP-address space is seen as one contiguous range of IP addresses (IPv4) from 0.0.0.0 to 255.255.255.255. The low parameter specifies the first IP-address in the range and the high parameter specifies the last IP-address in the range. The low parameter must be smaller or equal to the high parameter.

Protocol or IP-protocol

The term "protocol" is in this report used to identify which protocol an IP packet carries at the next higher protocol level. The entry is covering protocols specified in rfc 3232 [34] that defines higher level protocols and their numbers in the IP packet. Rfc 3232 refers to a database kept at www.iana.org [36]) here is a list of often used protocols and their related numbers in parenthesis. TCP(#6) see rfc 793 [22], UDP(#17) see rfc 768 [21], EGP(#8) see rfc 888 [23], IPv6-in-IP(#0) see rfc 2003 [28] etc. (see the full list in Appendix A)

Sub-protocol

The term "sub protocol" is used for protocol entries under each protocol e.g. ports for TCP, UDP defined by www.iana.org [37] and message types for ICMP also defined by www.iana.org [38] etc.

IP-session and IP-session-id

An IP-session is here defined as the flow of packets — as viewed on the wire — carrying information for the same TCP connection. A TCP connection is defined as communication between 2 ports. On the wire between the hosts communicating this will be seen as packets with the same source IP-address, source port number and source protocol (which is also TCP) and destination IP-address, protocol (which is TCP) and port number. In the NAT (and firewall) area we broaden this definition a bit so UDP and in some cases even ICMP messages-types can be used at the protocol part even if these protocols not formally has a "connection" state defined. Packets with identical protocol and port-numbers at a given IP-address often are related to e.g. the same higher level application, communicating with some other host IP-address. These 4 (6) parameters are during the report as a whole called the IP-session-id or the IP-session-id signature.

Local or internal addresses/network

"Local" or "internal" IP addresses are terms used for addresses not present on the Internet. The terms applies to other networks where certain addresses are not "defined" in the "overall" address plan for the network.

When private allocated network address ranges e.g. as defined by rfc 1597 [25] regarding the Internet. These IP-address ranges is not used at the public Internet as the Internet routing tables does not contain these addresses. Using address areas as defined in rfc 1597, you are sure that no other hosts on the Internet are using these IP-ranges. This way you will never have to face the problem of having the same IP address both at the internal network and at the Internet and thus avoiding problems with IP-address ranges at the Internet which are unreachable from your local IP-network behind your NAT.

Global or Public addresses/network

"Global" or "Public" IP addresses are in general IP addresses that belongs to the address plan of the Internet. The terms "Global" and "public" are often referring to the outside interface of a NAT device. See figure 3.1.

Inside network

The "inside" network is a network that is typically connected to the inside interface of a NAT device, or a network behind (or inside) a firewall. Figure 3.1 shows a typically "inside" network relative to the NAT device. Often the inside IP-subnet contains more defined IP-addresses seen from the NAT interfaces (LAN segment) than is seen from the outside IP-subnet.

Outside network

The "outside" network is a network that is typically connected to the outside interface of a NAT device or is located at the outside (in front of) a firewall. Figure 3.1 shows a typically "outside" network relative to the NAT device. Often the IP-subnet at the outside interface of a NAT device contains "fewer" defined IP-addresses compared to the inside NAT interface.

3.2 What is NAT

Network Address Translation (NAT) is most recently described by Internet Engineering Task Force (IETF)[1], in the RFC 2663 [31], in this way:

[1] IETF: www.ietf.org.

> "Network Address Translation is a method by which IP addresses are mapped from one realm to another, in an attempt to provide transparent routing to hosts. Traditionally NAT devices are used to connect an isolated address realm with private unregistered addresses to an external realm with globally unique registered addresses."

Which means that beside what most users are using NAT for — to connect network segments with (possible) independent address plans. NAT is intended to be used to translate addresses from one realm to another. Under normal use we do not often make the distinction of realms as we often use one e.g. Rfc 1597 IP-subnet together with addresses from the Internet address plan, and we do not have any address conflict as Rfc 1597 IP-ranges are deliberately not defined (by rfc 1597) at the Internet.

The term "Realm" often refers to kingdoms which can be understood as separate administrative domains. Here "Realm" is used to describe separate IP address domains (or kingdoms), e.g. IP address domains that is not necessary a part of the same IP (or the same overall IP) address plan. Different IP worlds - which do not has any relation to each other. No shared IP-addresses, no assumptions made of each other etc. NAT devices are intended to translate IP-addresses between different realms by having an interface in each of them.

NAT interfaces - naming conventions

Despite the above definition which defines NAT as translating IP addresses from one realm to another, we will use the terms "inside" and "outside" to describe the two interfaces of a nat device. These terms are referring to the fact that one interface often is connected to an internal (or private) network which is translated by the NAT device to an "outside" or public network. So when we name one interface the "inside" NAT interface we mean the side which has the "private" or the "many" IP addresses compared with the "outside" NAT interface which often has only one or a smaller number of IP addresses at the connected IP-subnet. NAT can be used as a one–to–one mapping between 2 IP networks – in which case the two interfaces act the same way. In most NAT setups one interface of the NAT has a larger address space configured than on the other NAT interface – and in these cases the above definitions applies.

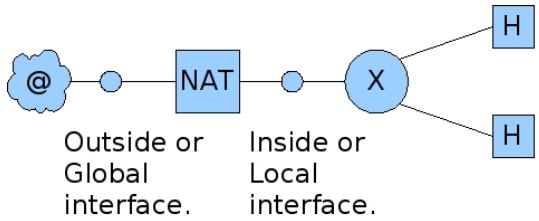


Figure 3.1: Global and Local NAT definitions. "X" is an IP router, "@" the Internet or similar global network, "H" is IP hosts.

A few more types of NAT

NAT is often used as a daily term for what really is NAPT. The precise difference is that NAT only translates IP addresses and NAPT translates both IP addresses and TCP and UDP port numbers between the inside and outside networks. During this report we will use the term NAT for both NAT and NAPT, when nothing more precise is mentioned.

When looking deeper into the nature of NAT we have to do further classification of NAT and defining a few new subclasses. NAT is actually an overall name for at least 3

ways of using the translation table that is the central element in all NAT devices. Here we categorize the NAT classes into the following classes: Static NAT, Dynamic NAT and Load Sharing NAT (LS-NAT). Each of these will be further described below.

Dynamic NAT

Dynamic nat is the most often used type of NAT. It is very often used when a private IP-network is connected to the Internet via a single public IP-address allowing all hosts at the private network to access services at the public Internet.

The "dynamic" part of the name refers to the fact that translation table entries are created when needed for packets arriving at the inside NAT interface and going towards the outside interface of the NAT device. This can be used in the "many" to "few" direction through the NAT device. When an incoming packet arrives at the inside NAT interface, and no NAT translation table entry exists for the IP-session to which the packet belongs, a new table entry is created. This entry is present in the table until removed from the translation table again.

Removal of dynamic created translation table entries can be done by several methods. The most often (but naive) used method is that entries are removed after a certain timeout when no packets for the connection has been seen for a predefined time. Other methods are tracking each packet flow and remove the entry when a TCP FIN packet is seen (the TCP Connection is closed), and use timeouts if sessions are never closed. For UDP which does not have a connected state, the only way is to use a timer to expire table entries (or apply interpretation for a higher protocol layer and get the information from there). This simple solution are often used as a simple way to provide NAT transparency for e.g. voice or video applications using UDP packet streams. The NAT is opened (a dynamic translation table entry is created) by sending UDP packets (from the client host at the inside interface of the NAT) through the NAT device to the stream server (located at the outside interface of the NAT) adding a translation rule for the UDP packets with the same session-id. Then the NAT translation rule exists when the service starts the stream of UDP packets e.g. Audio/Video media stream from the server located at the outside NAT interface allowing the traffic to pass through the NAT and reach the host at the inside network that opened (initiated) the "connection" from the inside.

Static NAT

Static NAT is often used when an internal (e.g. web server) IP-address must be reachable at an outside IP-address.

Static NAT is achieved by creating permanent entries in the NAT translation table. NAT translation table entries are inserted by the network administrator when the NAT device is configured. Removal of entries is done the same way.

A static NAT translation table entry is often used to create permanent translation table entries to allow hosts from the outside network to access services offered by hosts residing at the internal network.

As this has a one-to-many translation scheme and there is no way an algorithm (like in dynamic NAT) automatically can define translation table entries for packets that initially travels from the outside interface to the inside interface of the NAT.

Load Sharing NAT

Load Sharing NAT (LS-NAT) is another type of Static NAT which is described in rfc 2391 [32]. Load Sharing NAT is usable when we have a pool of internal servers configured to provide the (exact) same service set to client hosts. A number of translation table entries are configured by the NAT administrator one for each internal host offering the service. The table

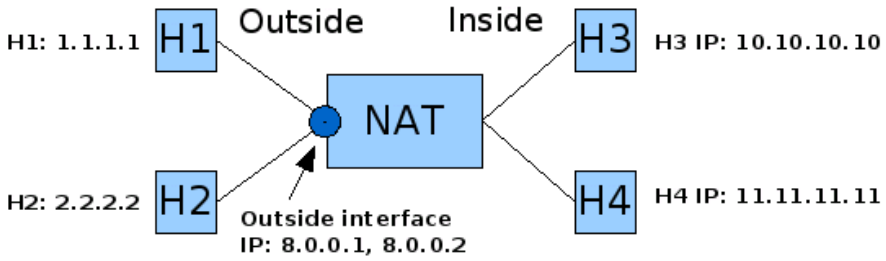
entries all share the same outside IP address and port number which we in the following will name the “outside-service IP-address”.

When a packet arrives at the ”outside-service IP-address” located at the outside NAT interface, its IP-session-id is looked up in the translation table if a match if found a translation to a specific host has already been setup for the IP-session. If no match is found this is a new IP-session which triggers the creation of a new translation table entry. The configured algorithm (e.g. round-robin or least-recent-used) is used to choose one of the configured servers for the IP-address and port number. When a server is selected a translation table entry is created for the new IP-session and the packets belonging to this IP-session is directed to this host as long the translation entry exists.

3.3 A detailed description of NAT

In this subsection we use the term “NAT” with its originally meaning and not with the meaning of NAT/NAPT.

The central element in a NAT device is the translation table. Each entry (row) in the table maps to one translation of inside and outside IP addresses. When an IP packet arrives to the NAT interface, a table lookup is performed to find a match for the TCP or UDP session the packet belongs to. The session is defined by the value of 4 parameters: IP source address, IP source protocol and port number, IP destination address and IP destination protocol and port number, where protocol is either TCP or (in many cases) UDP, but not all of these parameters are used in the lookup for every type of NAT. NAT most often uses only the inside and outside addresses. If a match is found, the packet is translated according to the matched entry, and the packet is forwarded to the other NAT interface. This is the same procedure for static as for dynamic NAT. The difference between these two NAT types is how a table entry is created and removed.



Entry	Outside	Inside
static 1	8.0.0.1	10.10.10.10
static 2	8.0.0.2	11.11.11.11

Figure 3.2: Schematic NAT overview. H1 and H2 are IP hosts located at the outside network, H3 and H4 are IP hosts located at the inside network. The NAT device translates IP addresses from the outside NAT interface (8.0.0.1 and 8.0.0.2) network into IP-addresses present at the inside network. The translation table has been pre-configured with 2 static NAT entries, which is seen in table.

In the following we will show a number of examples of what happens to packets when being translated by a static NAT device. We use the schematic layout from figure 3.2 which depicts 2 outside hosts “H1” and “H2” and 2 inside hosts “H3” and “H4” each with their configured IP-address shown next to the hosts. The translation table on the figure is used as the reference for the examples in the figures 3.3, 3.4, 3.5, 3.6 and 3.7.

When an IP-packet from hosts at the outside network hits the outside interface at IP-address 8.0.0.1 the translation table entry “static-1” is being used for the “destination IP-

address" field of the IP packet, and when a packet hits the outside interface at IP-address 8.0.0.2 the translation table entry "static-2" is being used for the "destination IP-address" field.

Figure 3.3 is an example of static nat, where the translation table has been configured by the network administrator when configuring the NAT device. The figure shows two versions of an IP packet sent from H1 to H3. The IP packet is sent from IP-address 1.1.1.1 to IP-address 8.0.0.1. When it reaches the NAT device on the outside interface a lookup is done for the IP-address in the destination field of the IP packet against the outside column in the translation table. This matches the entry "static-1" which has the IP-address 8.0.0.1 in the "outside" column and returns the matching inside IP-address 10.10.10.10 which is inserted into the destination IP-address field of the IP packet before being transmitted to the inside network by the inside NAT interface.

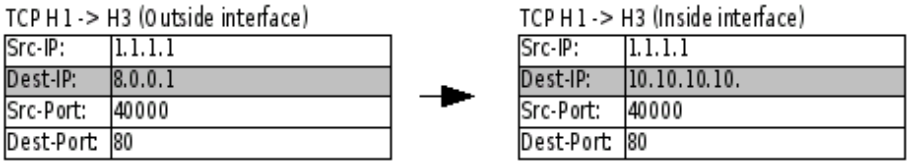


Figure 3.3: Shows an IP packet sent from host H1 to host H2. The two parts shows the difference in the IP packet from entry to exit of the NAT device. The left part shows the IP packet when passing the outside NAT interface and the right part shows the IP packet when passing the inside NAT interface into the inside network. The gray line shows the parameter changed by the NAT translation.

Figure 3.4 depicts a packet in the return direction for the same session as in figure 3.3. When the IP packet enters the inside interface of the nat device, the source IP-address of the IP-packet is looked up at the nat translation table against the inside column. A match is found which returns the outside IP-address 8.0.0.1 — which is inserted into the source IP-address field of the IP-packet, before being transmitted via the outside nat interface. This way the host H1 never sees anything else than the 8.0.0.1 IP-address in packets sent to this IP-address, despite they are going to H3 at IP-address 10.10.10.10.

The IP session described with the figures 3.3 and 3.4 shows that the NAT has performed an invisible translation from an outside network address to an inside network address. Application protocols at higher layers than TCP or UDP that contain IP-addresses of inside hosts which are not translated by the NAT this will disturb (or break) applications which rely on IP-addresses transmitted by protocols at higher levels to perform their tasks. The solution to this is to bring specific application layer gateways (ALG) into work. ALGs is a type of advanced NAT module which knows about specific protocols that need special treatment. ALG techniques is not a part of this project so we do not discuss this special type of NAT any further.

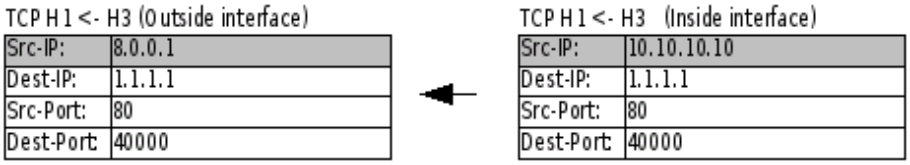


Figure 3.4: The IP packet returning from host H3 back to host H1. The left part shows the IP packet when exiting the outside NAT interface and the right part shows the IP packet when entering the inside NAT interface. The gray line shows the parameter changed by the NAT translation.

Figure 3.5 is the same scenario as the previous example, but this time another outside host H2 is sending an IP-packet to host H3. The destination IP-address of the IP packet is compared with the outside column in the translation table, and the entry "static-1" is again

matched. The IP-address 10.10.10.10 is inserted in the destination field of the IP packet before being send out of the inside nat interface.

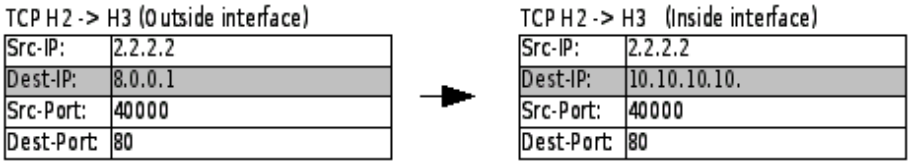


Figure 3.5: IP packet sent from host H2 to host H3. The left part shows the packet when entering the outside NAT interface and the right part shows the packet when exiting the inside NAT interface. The gray line shows the parameter changed by the NAT translation.

Figure 3.6 shows the IP packet returning from host H3 back to host H2. When entering the inside nat interface the IP-address in the source IP-address field is looked up at the inside column of the translation table and the IP-address 8.0.0.1 is returned and inserted into the source IP-address of the IP packet before being transmitted out of the outside nat interface.

The packets each contains all other data but the translation table entry for the inside host to be translated correctly - no saving of the outside host IP-address and port number is needed. This means that all sessions to the same outside NAT address are using (sharing) the same static translation table entry.

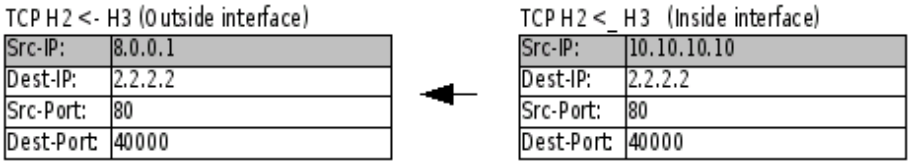


Figure 3.6: The IP packet returns from host H3 back to host H2. The left part shows the IP packet when exiting the outside NAT interface and the right part shows the IP packet when entering the inside NAT interface. The gray line shows the parameter changed by the NAT translation.

Figure 3.7 shows an IP packet sent from host H2 to host H4. This matches the “static-2” entry where the 8.0.0.2 IP-address is found in the outside column of the translation table. This returns the IP-address 11.11.11.11 which is inserted into the destination field of the IP packet before being transmitted out of the inside nat interface.

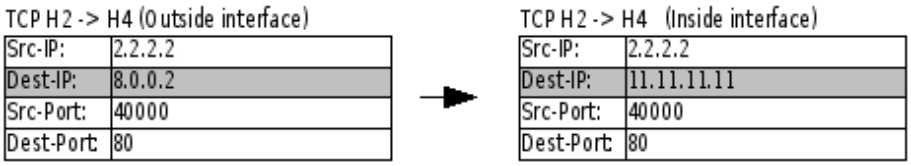


Figure 3.7: IP packet sent from host H2 to host H4. The left part shows the IP packet when entering the outside NAT interface and the right part shows the IP-packet when leaving the inside NAT interface. The gray line shows the parameter changed by the NAT translation.

We have now seen how NAT is able to hide the IP-addresses of hosts on internal networks. The examples here are all with static NAT, which typically is used when outside hosts (like H1 and H2) are going to access services on internal hosts (like H3 and H4).

Translation initiated from the inside to the outside NAT interface is treated a bit different as it is possible to automatically create translation table entries as traffics originate from the inside IP-subnets going toward IP-addresses located at the outside interface of the NAT. Traffic this way only needs an unallocated IP-address from the IP-range (or IP-address

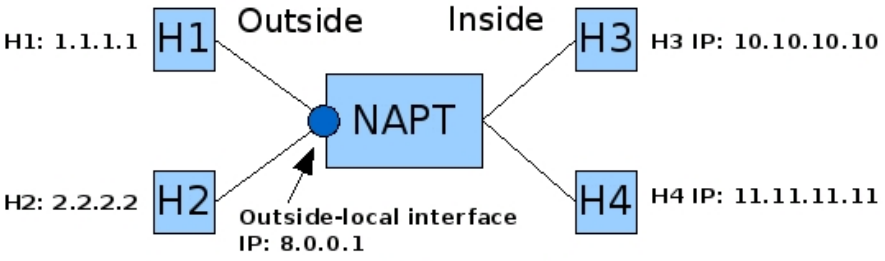
pool) of the outside NAT interface to pass. This allocation can be done automatically by the dynamic NAT function. The number of needed IP-addresses at the outside NAT interface is the number of simultaneous active inside hosts. One IP-address per simultaneous active inside host is needed at the outside NAT IP-address pool.

3.4 A detailed description of NAPT

Network Address and Port Translation (NAPT) is NAT with added port translation. Besides the IP-address NAPT also translates port numbers. NAPT is often used when a number of hosts on the inside NAPT interface is being translated to a single shared IP address (and assigned unique port numbers per session) at the outside NAPT interface. The problem that port translation solves is if two inside hosts have to share the same outside NAT interface IP-address, both are using the same source TCP port number — then the NAT needs to solve this conflict by translating one of the two conflicting IP-sessions (TCP) port numbers at the outside NAT interface IP-address to a unallocated (TCP) port number. When NAT is translating port numbers - it is formally called "Network Address and Port Translation" or short "NAPT".

The translation table shown in figure 3.8 contains data for the translations in the examples below. The format of each field of the table is IP-address:port-number. E.g. 8.0.0.1:80 means IP-address 8.0.0.1 port 80. The examples only shows TCP protocols and we do not differentiate between TCP and UDP in the examples — in the real world they are separate port ranges TCP and UDP are not sharing the same port number range.

The NAPT device shown in figure 3.8 has only one outside nat interface IP-address configured: 8.0.0.1, which is being used for all translations. The "static-1" entry has been configured by the administrator of the NAPT device to translate all incoming IP-packets at the outside NAPT interface matching IP-address 8.0.0.1 port number 80 to the inside IP-address 11.11.11.11 port number 8000. The 2 dynamic entries are created by the packets shown in the examples below.



Entry	Outside-source	Outside-local	Inside-local
static-3	-	8.0.0.1:80	11.11.11.11:8000
dynamic-1	1.1.1.1:80	8.0.0.1:40000	10.10.10.10:40000
dynamic-2	1.1.1.1:80	8.0.0.1:40001	11.11.11.11:40000

Figure 3.8: NAPT schematic overview. H1 and H2 are IP hosts located at the outside network. H3 and H4 are hosts located at the inside network. The NAPT device translates IP addresses from the inside network into IP addresses at the outside-local IP address (8.0.0.1) by the "dynamic" rules which are created when needed. Translation of traffic from outside hosts (H1 or H2) that originates at the outside NAT interface (8.0.0.1) will be translated by the "static-3" translation rule if its destination address matches port 80. The "static-3" translation rule has been pre-configured.

With the figures 3.9, 3.10, 3.11, 3.12, 3.13, and 3.14 we show detailed examples of NAPT translations. The gray shaded boxes in each figure shows the fields that are changed by the translation.

Figure 3.9 depicts an example of an IP packet that is sent from the inside host H3 to the outside host H1. When the IP packet reaches the inside NAPT interface a search at the translation table for the content of the source IP-address and the source port number of the IP-packet is performed against the "inside-local" column of the translation table. No match is found. As the NAPT device is configured as a dynamic NAPT device for packets traveling from the inside interface towards the outside interface — generation of a new translation table entry is triggered. The NAPT device chooses the 10.10.10.10:40000 to 8.0.0.1:40000 translation rule if it is available and inserts it into the translation table. In the example this is inserted as the "dynamic-1" row of the translation table together with the insertion of the destination IP-address and port number of the IP-packet data into the "outside-source" field of the translation table — to be able to identify which session the translation table entry belongs to when IP packets returns from the outside host H1. As the port numbers are the same between the inside and outside interfaces, no port numbers are (visibly) being translated and besides the automatic creation of the translation table entry everything works as in the static nat example above.

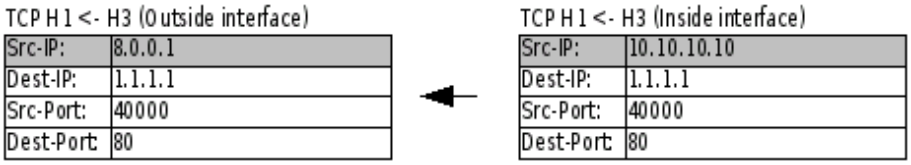


Figure 3.9: IP packet sent from host H3 to host H1. The left part is the IP-packet when it leaves the outside NAPT interface after translation and the right part is the IP packet when is enters the inside NAPT interface. The gray marked areas shows the parameters changed by the NAPT translation.

Figure 3.10 shows a return IP packet for the same session as figure 3.9. The IP-session-id signature[2] of the IP packet is looked up against the "outside-local" and the "outside-source" columns of the translation table and a match with the "dynamic-1" entry is found. The value 10.10.10.10:40000 is returned from the translation table and inserted into the destination IP-address and destination port fields of the IP packet before it is transmitted out via the inside NAPT interface.

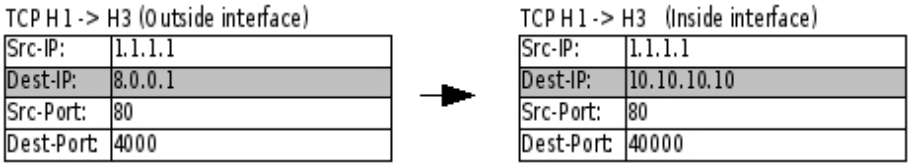


Figure 3.10: IP packet returns from host H1 back to host H3. The left part shows the IP packet when entering the outside NAPT interface and the right part shows the IP packet when leaving the inside NAPT interface after being translated by the NAPT translation entry "dynamic-1" in figure 3.8. The gray marked areas shows the parameters changed by the NAPT translation.

Figure 3.11 shows an IP packet from the inside host H4 being sent to the outside host H1. When the IP packet reaches the inside NAPT interface the source IP-address and source port number is looked up at the inside column of the translation table. No match is found. As dynamic NAPT is configured for this direction trough the NAPT the creation of a new translation table entry is triggered. This time the 8.0.0.1:40000 outside address is occupied by the session established in figure 3.9 and figure 3.10 (assuming the session is still existing). An available outside IP-address and port number pair has to be found. Here 8.0.0.1:40001 is selected and inserted into the translation table as the "dynamic-2" entry with the destination IP-address and port number of the IP-packet to identify return packets

[2] See the definition of the IP-session-id in section 3.1 page 17.

belonging to the same IP-session (outside-source: 1.1.1.1:80, outside-local: 8.0.0.1:40001, inside-local: 11.11.11.11:40000). The content of the source IP-address and source port number and protocol of the IP packet is exchanged with these values before being transmitted out of the outside NAPT interface towards host H1.

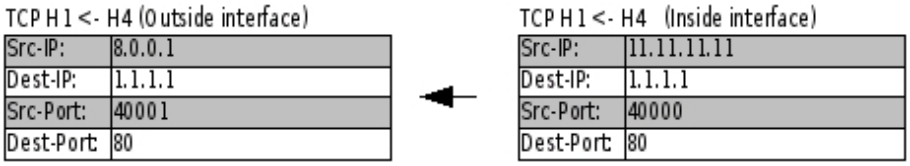


Figure 3.11: IP packet originating from host H4 sent to host H1. The left part shows the IP packet when leaving the outside NAPT interface after translation and the right part shows the IP packet when entering the inside NAPT interface. The gray areas shows the parameters changed by the NAPT translation.

Figure 3.12 shows a return packet from the same session as in figure 3.11 above. When the IP packet reaches the outside NAPT interface its IP-session-id (source: 1.1.1.1:80, destination: 8.0.0.1:40001) is looked up in the outside-source and outside-local column of the translation table and a match is found at the "dynamic-2" entry. The IP-address 11.11.11.11 port number 40000 is returned and inserted at the destination IP-address and port number fields of the IP packet, before being transmitted out of the inside NAPT interface towards host H4.

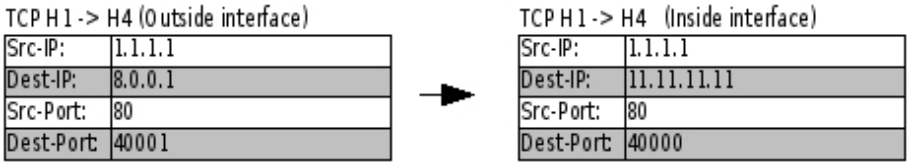


Figure 3.12: IP packet returning from host H1 back to host H4. The left part shows the IP packet when entering the outside NAPT interface and the right part shows the IP packet when leaving the inside NAPT interface. The gray areas shows the parts of the packet that has been changed by the NAPT translation.

We have now seen how NAPT is able to hide a number of hosts at an internal network behind a single IP-address by translation of the IP-addresses and port numbers of the internal hosts into outside NAPT interface IP-addresses and port numbers.

In figure 3.13 we show a static NAPT translation using the "static-3" entry of figure 3.8. This will translate IP packets arriving at the outside NAPT interface destined for 8.0.0.1:80 to the host H4 port 8000 (11.11.11.11:8000) where a service (e.g. a http server listening at TCP port 8000) is expected to be answering. This is used when outside hosts are going to access services on hosts located at the inside network. When the IP packet is reaching the outside NAPT interface the IP-session-id is looked up against the translation table. As the "static-3" entry has a wild card (or don't care marker) in the "outside-source" column the 'static-3" entry is found. The translation pair. 11.11.11.11:8000 is returned and inserted into the destination field of the IP packet before being transmitted out of the inside NAPT interface towards the host H4. As the entry is static (permanent existing) the NAT device does not have to follow the session creation and termination hence there is no need to save the content of the "outside-source" field as these are carried in each IP packet anyway (in this example the 1.1.1.1:30000 pair).

It is not in all circumstances that IP packets sent to the outside interfaces are translated by the static NAT. In the next example we show an example where this is not the case.

In figure 3.14 we see a packet sent from host H1 to the 8.0.0.1:25 IP-address and port number of the outside NAPT interface. The IP-session-id of the IP packet is looked up

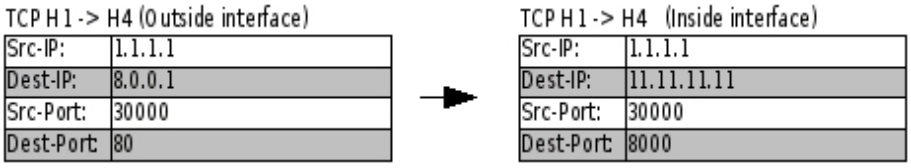


Figure 3.13: Static translation of IP packet from host H1 to host H4. The left part shows the IP packet when entering the outside NAPT interface and the right part shows the IP packet when leaving the inside NAPT interface. The gray areas shows the parameters changed by the NAPT translation.

against the "outside-source" and "outside-local" column of the translation table, and no match is found. This means that there is no translation possible for this packet and no forwarding onto the inside NAPT interface will happen. The packet is silently dropped (or an ICMP error packet might be sent back depending of the implementation).

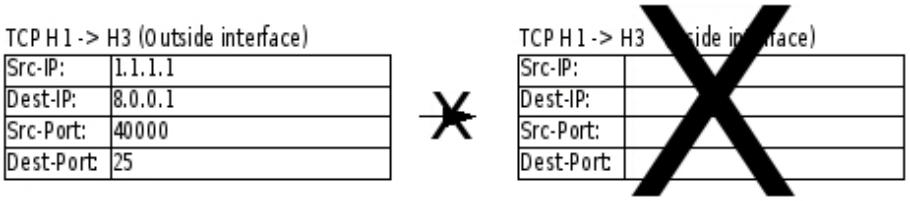


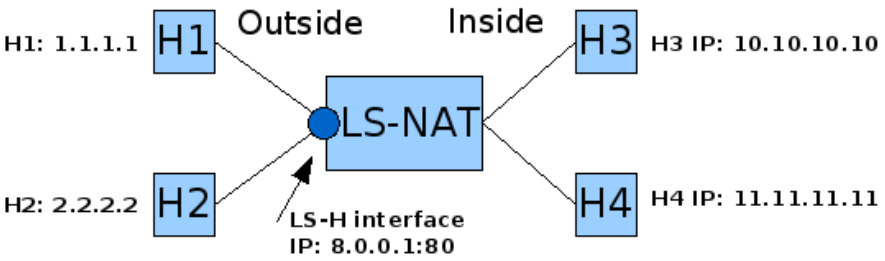
Figure 3.14: IP packet from host H1 to host H3 with no matching translation table entry. The left side shows the IP packet when entering the outside NAPT interface and the right part shows the packet when leaving the inside NAPT interface. As the packet is not forwarded onto the inside NAPT interface - the packet is crossed out.

3.5 A detailed description of LS-NAT

Load Sharing NAT (LS-NAT) is the third type of NAT module uses that we describe thoroughly in this report. LS-NAT is used as a simple way of distributing incoming traffic for a service between a group of hosts located at the inside LS-NAT interface all serving the exact same content. If we as an example think of a web server, then all the content that the web service has must be available at each web server that participate in the load sharing group for a given service. The main idea is that each TCP connection is serviced by the same host during its lifetime - thus when 2 servers are available each only has half of the load. Alternative (and more ideal for a web service) it would be ideal if all TCP connections from a users web session is serviced from the same internal web server during the session lifetime. As the LS-NAT device only know about the IP packets flowing to and from the load shared servers, it is not by any mean an optimal load sharing for sessions, but for simpler applications e.g. photos or graphics for advertising it is really useful.

LS-NAT devices are a specialized version of a static NAPT module where the service IP-addresses of the participating servers are defined in a load sharing group at configuration time. When an incoming connection arrives the LS-NAT module selects one of the servers in the load sharing group to service the current request. The selection of a server follows an algorithm defined at configuration time. Often used load sharing algorithms are round-robin and "least recently used" server. More advanced algorithms like e.g. least loaded server requires some means to measure the load of the servers participating in the load

sharing group. This is a more advanced LS-NAT setup which we define as outside the area for this project.



Entry	Outside-source	Outside-local	Inside-local
snat-H3	-	8.0.0.1:80	10.10.10.10:8000
snat-H4	-	8.0.0.1:80	11.11.11.11:8000
session-H1	1.1.1.1:40000	8.0.0.1:80	10.10.10.10:8000
session-H2	2.2.2.2:40000	8.0.0.1:80	11.11.11.11:8000

Figure 3.15: Load Sharing NAT (LS-NAT) schematic overview. Host H1 and H2 are client hosts of the published service LS-H at IP-address 8.0.0.1 port 80. Hosts H3 and H4 are member of the same load sharing group and offers the same service. Ideally each server serve half of the load.

In the following we will describe a few examples of how a LS-NAT can work. We emphasize that there is no standard way of operation for LS-NAT, so there is a wide area for developers to design solutions for specific needs of each implementation e.g. in the area of identifying sessions, what algorithms to use and how to get the relevant data for the algorithm in real time. A naive detection algorithm for sessions is to use only the IP number of the client hosts and not the port number of the packets when mapping packets to inside servers. At a first glance this solution seems to have the drawback that if a larger number of users are located behind a NAPT device then all their traffic will be assigned to the same server and no load sharing will occur for them. This is not the case as the NAPT device will use different port addresses at the outside interface for each connection to the load shared service address. We will not go any further into discussion of which parameters to use for identifying client traffic. In all our examples here we are using both IP-address and port number of the client connections for the identification.

Removal of "session-" entries of the LS-NAT translation table can also be done in a number of ways this is also one of the places where the actual design is highly decided by the implementor. An easy but naive approach is to only remove dynamic entries after timeouts to keep the same users using the same servers for the longest time and across multiple TCP connections. This approach can be helping e.g. web services to get a better load sharing efficiency, by reusing the already recently used items again for the same user. Web services uses a large number of relative short-lived TCP connections typically one for each item at every web-page. A "user" bound to the same load-sharing host for the longest time will utilize the servers internal caching mechanisms. For example files already cached in the in-memory disk-cache will be utilized when the file is retrieved more times within short intervals during a user session lifetime.

Figure 3.15 shows a schematic LS-NAT setup and an example LS-NAT translation table. The contents of the table is used and explained in the examples below.

The schematic diagram shown in figure 3.15 shows two outside client hosts: H1 and H2 that are accessing the service offered at the load sharing outside address 8.0.0.1:80, which we label "LS-H" (for LS-Host) in the descriptions. The LS-H is the published outside IP-address for the service offered by the load sharing group. The load sharing group consists of host "H3" and host "H4" located at the inside interface of the LS-NAT.

During the next pages we will provide a detailed description of the principles of LS-

NAT. We will describe 4 packets traveling through the LS-NAT and explain what happens with them during the translation process. The translation table shown in figure 3.15 contains data from the examples described below.

Figure 3.16 shows an IP-packet originating from host "H1" toward the "LS-H" load sharing IP-address. When the IP packet reaches the LS-NAT its destination IP-address and port number together with the source IP-address and port number are looked up against the LS-NAT translation table. First the dynamic entries list (in figure 3.15 it is the entries name starting with "session-") is searched. If no match is found, the IP packet belongs to a unknown session which then triggers the creation of a new "session-" entry in the translation table with the IP-session-id matching the triggering packet.

A new "session-" entry is created by first searching the configured load sharing groups in the translation table. This search is performed by a search for the destination IP-Address and port number against the "outside-local" field of the translation table. A Match is found at line "lsnat-H3" and "lsnat-H4". The algorithm (here round-robin is configured) selects the first entry which is "lsnat-H3" which returns the IP-address and port number 10.10.10.10:8000 that replaces the destination fields of the IP packet. The algorithm (here round-robin is configured) selects the first entry which is "lsnat-H3" which returns the IP-address and port number 10.10.10.10:8000. These parameters replaces the content of the destination fields of the IP packet. Hereafter the IP packet is sent to host H3 via the inside LS-NAT interface. The new entry is inserted into the translation table labeled "session-H1" with the content outside-source: 1.1.1.1:40000, outside-local: 8.0.0.1:80, inside-local: 10.10.10.10:8000.

The next packet with an IP-session-signature matching the packet in figure 3.16 will match the new entry "session-H1" in the translation table and be directed to host H3 by the LS-NAT as it is identified as belonging to the same load-sharing session.

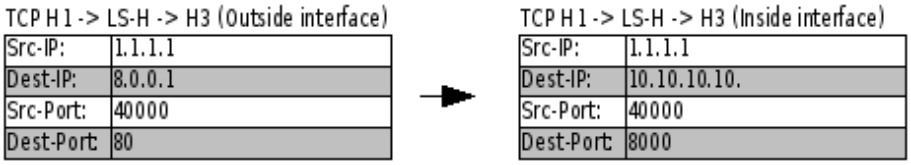


Figure 3.16: IP packet from host H1 sent to the LS-H public service address. The left part shows the IP packet sent from the client host H1. The right part shows the LS-NAT translated packet sent from the LS-NAT inside interface towards host H3. The gray areas shows which parameters the LS-NAT has changed during the translation.

Figure 3.17 shows the IP packet returning from host H3 back to host H1. When the packet arrives at the inside interface of the LS-NAT its source IP-address and port number is looked up at the translation table against the "inside-local" column. The "session-H1" row has a match. The content of the field "outside-source" is returned (1.1.1.1:40000) and inserted into the destination IP-address and destination port number fields of the IP packet before it is sent to host H1 via the outside LS-NAT interface.

Figure 3.18 shows an IP-packet from a new client host H2, that is accessing the service offered at the LS-H address. When the IP packet reaches the outside LS-NAT interface the same happens as in the figure 3.16 example. The source IP-address and port number together with the destination IP-address and port number are looked up against the translation table "session-" entries and no match is found. The LS-NAT now identifies the packet as belonging to an unknown TCP connection and triggers the generation of a new "session-" entry in the translation table.

This is done in much the same way as the example shown in figure 3.16 at page 28. The new entry is generated by looking up the destination IP-address and port number of the IP packet against the "outside-local" column at the translation table "lsnat-" entries. The list of hosts in the load sharing group for the LS-H service returns the IP-address

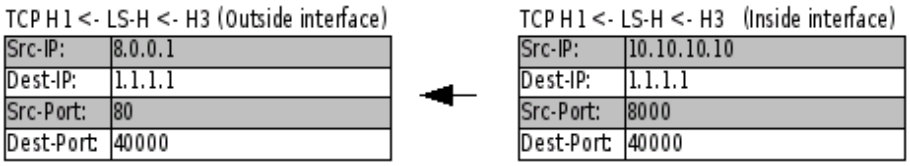


Figure 3.17: IP packet from host H1 sent back to the H1 client. The IP packet is translated to appear from the LS-H public service IP-address. The left part shows the IP packet sent from the LS-H interface at the LS-NAT device. The right part shows the packet returned from host H3 back to H1 when it enters the inside interface of the LS-NAT device. The gray areas shows which parameters the LS-NAT has changed during the translation.

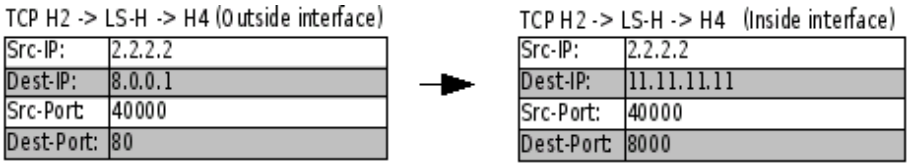


Figure 3.18: IP packet from host H2 sent to the LS-H public service address and translated by the LS-NAT device to host H4. The left part shows the IP packet sent from the client host H2 to the LS-H address of the LS-NAT outside interface. The right part shows the LS-NAT translated packet sent from the LS-NAT inside interface towards host H4. The gray areas shows the parameters changed by the LS-NAT translation.

and port numbers of host H3 and H4. As the round-robin algorithm returned host H3 address and port number (10.10.10.10:8000) the last time it was called this time it returns the IP-address and port number for host H4 (11.11.11.11:8000), which are inserted into the destination fields of the IP packet before it is sent to host H4 via the inside LS-NAT interface of the LS-NAT device. At the same time the new session entry is inserted into the translation table labeled “session-H2” with the parameters: outside-source: 2.2.2.2:40000, outside-local: 8.0.0.1:80, inside-local: 11.11.11.11:8000 to ensure that the following TCP packets from this host and port number is sent to the same load sharing server (H4).

Figure 3.19 shows the return packet which is the reply for the packet shown in figure 3.18 at page 29. When the IP packet reaches the inside LS-NAT interface its source and destination IP-address and port numbers are looked up in the translation table, a match is found at the entry labeled “session-h2” the source IP-address and port number is swapped with the values from the table entry column “outside-local” (8.0.0.1:80) which is the LS-H value of the public service IP-address and port number.

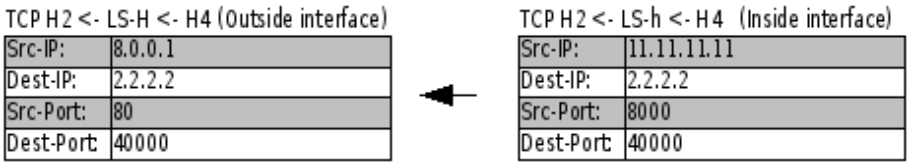


Figure 3.19: IP packet from host H4 sent back to the H2 client. The IP packet is translated by the LS-NAT to appear as arriving from the LS-H public service IP-address. The left part shows the IP packet sent from the LS-H interface at the LS-NAT device back to host H2. The right part shows the packet returned from host H4 back to H2. The gray areas shows which parameters the LS-NAT has changed in the IP packet during translation.

Summary

We have now described the vocabulary around NAT and IP terminology and used this in the description of the various flavors of NAT. We have described the various flavors of NAT: NAT which is IP address translation (only), NAPT which is IP address and port translation, dynamic and static NAT/NAPT and finally the Load Sharing NAT (LS-NAT). All types of NAT has been described thoroughly with a number of examples. This has given us a good and precise understanding of how the different types and functions of NAT is working which we can use in the evaluation of the commercial and open source derived NAT types and as a basis for defining the requirements for a configuration language for the XORP NAT module.

Chapter 4

Analysis of current NAT implementations

4.1 Analyzing existing NAT implementations

In this chapter we will do a short analysis of NAT implementations from commercial and open source implementations. We have analyzed a few commercial NAT implementations: Cisco and Juniper and the FreeBSD open-source implementation "natd". During the analysis we will focus on the following features of each analyzed implementation of NAT:

- Where the NAT module are located compared to the routing function and the network interfaces of each system.
- Is the NAT module able to handle more than two NAT interfaces (e.g. an outside and an inside interface).
- The configuration language and vocabulary used in each implementation. It will not be a thorough analysis, but if interesting elements springs into our eyes we will describe these.
- We will try to identify (obvious) advantages or limitations of each of the analyzed implementations.

We are analyzing how the selected well-known implementations handle these properties, and acquiring knowledge to design our own configuration language.

4.2 The Cisco NAT implementation

The following description refer to the Cisco configuration example in figure 4.2 at page 33. Cisco has two sided NAT and uses the terms "outside" and "inside" to name the two interfaces of the NAT module. Interfaces are marked in their configuration clause with either `inside` or `outside` if they belong to any side of a nat function block. Named NAT pools defines the global IP-addresses used by the NAT module. Access control lists (ACL) is used to define which IP-packets are going to be treated by the NAT module (Cisco uses the configuration term `access-list` to define ACLs). Only the IP packets selected by the ACL for the nat module is translated, all other IP packets passing a NAT tagged interface are not translated. This way Cisco has a way to define exact which packets are translated by the NAT module. The NAT function is defined by the `ip nat inside source` command, which means that any packets received at the `inside` (labeled) interface will have their source IP-address translated to an address in the named pool. The `overload` parameter

of the `ip nat inside source` configuration line enables NAPT (port translation) if the `overload` parameter is not included in the line only NAT (IP-address translation) is performed.

When packets passes an interface marked as either `inside` or `outside` and fits the Access Control List (ACL) triggering one of the configured NAT modules in the router, the packets are sent trough the triggered NAT device. Cisco describes their NAT module placement in the router with these words:

> "Packets traveling from the inside side to the outside side of the NAT function has the NAT function translating their source address and port-numbers after the routing function.
>
> Packets traveling from the outside to the inside side of the nat box will pass the NAT function before the routing function is applied to the packet. Figure 4.1 shows the Cisco NAT location related to the interfaces (if) and IP router module (marked with an "X")."

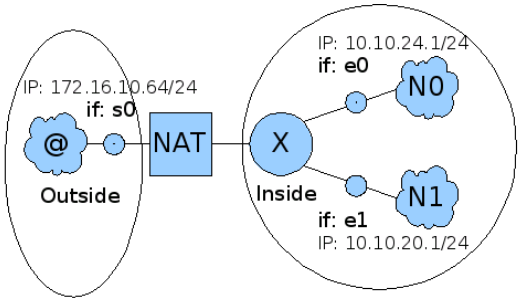


Figure 4.1: Cisco schematic NAT diagram. "X" is the router process, "N0" and "N1" are IP subnets, "if" marks the possible locations of Cisco interfaces, "@" is the Internet or an outside IP network. The circles marks the "inside" and "outside" areas for the NAT definitions. The IP-addresses and interface names matches the Cisco configuration example shown in figure 4.2 example.

Cisco supports dynamic and static NAT. Load sharing NAT (LS-NAT) is only for more advanced Cisco boxes specially designed with load sharing purposes in mind (Cisco calls these Content Switches) The Cisco NAT configuration example in figure 4.2 is from the Cisco technical document "Configuring Network Address Translation – Getting started" [39].

The example below shows a typical Cisco NAT setup, where hosts on the inside network at interfaces 'e0' and 'e1' are accessing hosts located at the public (outside) network via the "s0" interface. Only the first 31 IP-addresses from each network is translated by the NAT this is limited by the ACL filters defined at the `access-list` configuration commands.

Translation table timeouts

Cisco has a number of commands to control the timeouts of translation table entries. For NAT (IP-address) translation it is set with this command:

```
ip nat translation timeout <seconds>
```

Dynamic NAT translations time out after a period of non-use. When port translation is not configured, translation entries time out after 24 hours. This time can be adjusted with the above command or with the following variations:

```
ip nat translation udp-timeout <seconds>
ip nat translation dns-timeout <seconds>
ip nat translation tcp-timeout <seconds>
ip nat translation finrst-timeout <seconds>
```

```
!--- Defines Ethernet 0 with an IP address and as a NAT inside interface.
interface ethernet 0
 ip address 10.10.10.1 255.255.255.0
 ip nat inside

!--- Defines Ethernet 1 with an IP address and as a NAT inside interface.
interface ethernet 1
 ip address 10.10.20.1 255.255.255.0
 ip nat inside

!--- Defines serial 0 with an IP address and as a NAT outside interface.
interface serial 0
 ip address 172.16.10.64 255.255.255.0
 ip nat outside

!--- Defines a NAT pool named ovrld with a range of a single IP
!--- address, 172.16.10.1.
ip nat pool ovrld 172.16.10.1 172.16.10.1 prefix 24

!--- Indicates that any packets received on the inside interface that
!--- are permitted by access-list 7 will have the source address
!--- translated to an address out of the NAT pool named ovrld.
!--- Translations will be overloaded which will allow multiple inside
!--- devices to be translated to the same valid IP address.
ip nat inside source list 7 pool ovrld overload

!--- Access-list 7 permits packets with source addresses ranging from
!--- 10.10.10.0 through 10.10.10.31 and 10.10.20.0 through 10.10.20.31.
access-list 7 permit 10.10.10.0 0.0.0.31
access-list 7 permit 10.10.20.0 0.0.0.31
```

Figure 4.2: Cisco NAT configuration example showing a complete Cisco NAT dynamic configuration that allows hosts from the first 30 IP-addresses from each inside network (interfaces e0 and e1) to be translated to the outside interface (s0). The example is from the Cisco technical document "Configuring Network Address Translation – Getting Started" [39]

When port translation (NAPT) is configured, it is possible to acquire a finer control over translation entry timeouts, because each entry contains more context about the traffic using it. The Cisco defaults are: Non-DNS UDP translations time out after 5 minutes; DNS times out in 1 minute. TCP translations time out after 24 hours, unless a RST or FIN is seen on the stream (with the same IP-session-id), in which case it times out in 1 minute.

4.3 Juniper NAT implementation

Juniper networks IP-router operating system is called JUNOS. It has a significant different configuration structure than Cisco. JUNOS has a tree-like structure with configuration clauses (or statements) defined for each part of the configuration. Each location in the configuration tree is named by listing the nodes when traversing the nodes of the configuration tree from the root to the location being described. The NAT configuration statements are located below the position `services nat` in the JUNOS configuration-tree.

The whole `services nat` configuration tree is shown in figure 4.3 at page 35. We provide a short explanation of the most important details here. When Juniper use names as "source" and "destination" in their NAT configuration language, they refer to the relevant fields of the IP-packet that is being translated by the configuration. The named nodes in the list below starts at the `services` offset at the configuration tree. To help reading and understanding the structure of the language we are using the same indentation as the language itself:

`nat pool <nat-pool-name>` defines named IP-address definitions that can be referenced later in the configuration. A definition can be a single IP-address, an IP-subnet or an IP-address range. It can include a protocol and port specification part.

nat rule <rule-name> Each NAT rule consists of a set of named `terms`, similar to a firewall filter and a `match-direction <direction>` statement that defines the direction of IP-packets the NAT rule acts upon. Valid `<direction>` arguments are: `input`, `output` and `input-output`.

term <term-name> clause defines the NAT properties of the named NAT term. The `term` clause has the following sub clauses: `from` and `then translated`.

from The `from` statement specifies the match conditions that are included and excluded in the translated set of IP-addresses and ports. The `from` clause has a row of sub entries which all defines IP address and port ranges in various ways. The `applications` and `application-sets` defines IP-address and port definitions for applications or sets of applications, `destination-address` and `source-address` clauses defines IP-addresses or IP-subnets. All the definitions under the `from` clause has the function of ACLs for IP packets that are being translated by the `then translated` definition in th same NAT `rule`.

then translated The `then translated` statement specifies the actions and action modifiers to be performed by the NAT module. The `then translated` clause has 3 sub clauses:

source-pool <nat-pool-name>

destination-pool <nat-pool-name>

translation-type Specifies what type of NAT is used for source or destination traffic translation. It can have the following values:

destination static Implements address translation for destination traffic without port mapping. This translation type requires the definition of a named `destination-pool` with only one IP-address and no port defined.

source static Implement address translation for source traffic without port translation. This translation type requires the definition of a named `source-pool` to be defined. The referenced pool must contain exactly one address or prefix and no port configuration. Exactly one source-address value must be included in the `from` statement in the same `term <term-name>` statement as the `source static` statement. If this element is a IP-subnet the size must be less than or equal to the pool IP-subnet size.

source dynamic Implements address translation for source traffic with port translation (NAPT). A named `source-pool` definition must be defined. The referenced pool must include a port or address definition. This translation type supports translation of a large IP address range to a smaller size pool.

JUNOS supports only NAT modules with 2 interfaces, and it seems that the implemented configuration syntax has the normal NAT and NAPT feature (rich) set as described in the rfc 2663 [31]. ACLs are configured with the `from` statement and has the normal set of configuration features. JUNOS has the interesting feature that the configuration vocabulary `source` and `destination` keywords refers to the IP-addresses of IP packets e.g. source refers to the source IP-address field. JUNOS does not seem to have LS-NAT functionality implemented in connection with the normal NAT functionality. We have not been able to detect any information about timeouts of NAT translation table entries for JUNOS in the JUNOS documentation as well as we did not find any information about the precise location of the NAT module relative to the routing process and the interfaces.

```
nat {
     pool nat-pool-name {
       address (address | address-range low value high value | prefix);
       port (automatic | range low value high value);
     }
     rule rule-name {
       match-direction (input | output | input-output);
       term term-name {
          from {
             applications [ application-names ];
             application-sets [ set-names ];
             destination-address (address | prefix);
             source-address (address | prefix);
          }
          then {
             translated {
                destination-pool nat-pool-name;
                source-pool nat-pool-name;
                translation-type (destination type | source type);
             }
             syslog;
          }
       }
     }
     rule-set rule-set-name {
       [ rule rule-names ];
     }
}
```

Figure 4.3: Juniper JUNOS NAT configuration overview. The example is from the JUNOS technical documentation [40].

We had access to the JUNOS release 7.5 documentation [40] when analyzing this. It seems that the newer JUNOS release 8.2 has a few minor features added to the NAT configuration language - but there is still no information about translation table entry timeouts for various protocol types or in general.

4.4 FreeBSD NAT implementation

The FreeBSD NAT implementation lives in the "natd" user land program. The IP packets are routed from the kernel via the IP DIVERT kernel module, which routes traffic that matches a given property (configured with the ipfw divert "rule action") to the user land "natd" program. After the packets are translated by the "natd" module, packets are sent back to kernel space through the IP DIVERT interface. The IP-divert command can include ACL expressions that will select the IP packets that is passed to the NAT module.

The FreeBSD "natd" program can only handle a smaller set of IP addresses, IP-ranges or one IP subnet at a time. The configuration can take an interface name as the global side interface configuration. Several static translations can be set up and more complex configurations can be loaded from a file.

We will describe a few of the configuration details of `natd` in the following paragraphs. All configuration examples are written as command line arguments to the `/usr/sbin/natd` command.

The argument below provides static NAT functionality.

```
-redirect_address localIP publicIP

-redirect_port proto targetIP:targetPORT[-targetPORT]
                [aliasIP:]aliasPORT[-aliasPORT]
                [remoteIP[:remotePORT[-remotePORT]]]
```

According to the manual page If the `-redirect_port` example parameter called "targetIP") is supplied with more comma separated IP-addresses it will become a LS-NAT setup. "targetIP" and "targetPORT" refers to the servers that are load-shared. "remoteIP" and "remotePORT" are the outside IP address and port of the load-shared service. In many

aspects the function performed is similar to static NAT but by adding an algorithm that controls the distribution of new sessions to one of the configured servers the "natd" NAT is able to function as a load sharing device too. The syntax for LS-NAT is shown below:

```
-redirect_port proto
               targetIP:targetPORT[,targetIP:targetPORT[,...]]
               [aliasIP:]aliasPORT [remoteIP[:remotePORT]]

-redirect_address localIP[,localIP[,...]] publicIP
```

Dynamic NAT is setup like this with global IP address derived from the name of the interface, all other addresses at the host are by default defined as "inside" IP-addresses. The following option defines dynamic NAT with the global IP-address at the named interface:

```
-interface | -n <interface>
```

If the interface might change its assigned IP-address during operation (e.g. if the DHCP server assigns a new IP-address for the interface during lifetime of the "natd" operation) the `-dynamic` option must be given too.

Dynamic NAT with a configured global IP address. Only one of the options `-dynamic` or `-alias_address` can be used at the same time.

```
-alias_address | -a <address>
```

Natd supports a feature that let it try to use the same (TCP/UDP) port on both sides of a dynamic translation. By not changing the port number during the NAT translation protocols such as Remote Procedure Calls (RPC) has a better chance of working. This option is enabled by setting the following option in the configuration command:

```
-same_ports  | -n
```

Natd has support for interfacing with external proxy server processes running at the host which we do not address further during this project.

The FreeBSD "natd" can only handle a small number of global address definitions. We will have to test weather more "natd" processes can be running at the same time at the same global interface or at the host with different global interfaces defined at the command line.

The FreeBSD "natd" NAT module has a wide range of configuration possibilities. It supports NAT, NAPT and LS-NAT operation. Has support for two interfaces, can both use named interfaces with their existing (dynamic) IP configuration and static assigned IP-addresses as configuration options for the outside interface. ACLs is supported via the IP divert kernel module where normal "ipfw" firewall configuration commands can select the IP packets to be translated by the NAT module.

The "natd" NAT module has a smaller and compact configuration "language" and the NAT module is located between the interface that has the "outside" NAT interface and the hosts routing process (just as shown at the Cisco schematic model at figure 4.1 at page 32.)

4.5 Conclusion

We have now evaluated 3 different implementations of NAT, 2 commercial which only has static and dynamic functionality, and an open source version that also supports LS-NAT. The two commercial versions has quite the same set of supported functionality and the FreeBSD "natd" module seems to support the same set of concurrent features and translation schemes. As the FreeBSD "natd" module only has one instance the more complex configurations and multiple rules (or NAT instances) possible in both Cisco and Juniper NAT configuration languages is not possible in the FreeBSD "natd" module.

Cisco uses the terms "inside" and "outside" to identify (or bind) NAT interfaces to router interfaces , Juniper JUNOS NAT configuration uses "source" and "destination" terms

which is merely termed at the names of fields in the IP packets being translated. JUNOS has no mention of router interfaces in their configuration language, all configuration is about matching IP-addresses - no named interfaces is used. FreeBSD “natd” NAT module does uses the name of the outside interfaces if it has a dynamic IP configuration (e.g DHCP assigned IP addresses).

Chapter 5

A configuration language for XORP NAT

Introduction

Today XORP has no configuration language support for NAT. During this chapter we will analyze the various requirements for configuring a NAT device, search for a suitable configuration model and develop an configuration language for the NAT module according to the model.

Our goal is to design a general applicable configuration language for NAT, as this will hopefully ease the configuration language support of new host OS native NAT modules without having to change central parts of the configuration language.

As the FreeBSD network stack does not currently have realms implemented in its IP stack, the following discussion of realms is somehow theoretic but as this might change someday we will consider this possibility anyway. The host-OSes we know about today does not support realms, so the practical implementation might be different than suggested here.

5.1 Defining a model and a configuration language

Implementing a NAT device into XORP requires a configuration language able to describe the desired configuration of the NAT. We are pursuing a configuration file syntax which is able to express the various setups a NAT module can have. When we create a general applicable configuration language we will end up with a NAT configuration syntax that can express functionality that is not achievable by all the possible host-OS NAT modules supported, as some of these are more limited in their feature set than others.

After having analyzed configuration details of various open source and commercial NAT products, we evaluate the various elements and ideas found with respect to creating a configuration language for the XORP NAT module.

The general assumption is that all possible NAT functionality can be implemented by the Click packet-rewrite module, and a subset of all possible NAT functionality can be implemented with the native NAT modules of FreeBSD and Linux.

From a first hand perspective it seams that various native NAT implementations (FreeBSD, Juniper and Cisco) all have quite some differences between what is possible and the notation and vocabulary used in the configuration languages for these modules.

During the analyzing of existing NAT implementations we learned that a number of parameters has to be specified to get the possible NAT setups that each product is able to achieve. We have identified at least 3 types of NAT functionality: Static NAT, dynamic

NAT and Load-Sharing NAT. Each of these has different parameters in their configuration: Inside and outside NAT IP-address definitions, some even requires TCP or UDP ports to be specified as a part of their setup.

The language we design must be able to specify either a fixed IP or an named interface which has a pre-configured IP-address e.g. from a DHCP client assignment before XORP is started.

The NAT module is usually viewed as a 2 sided module. Usually we name the 2 sides the “inside” and the “outside” interface. For most uses NAT is only using 2 sides and as far as we know most existing NAT modules are implemented this way.

During the initial phases of designing the configuration language, we have been evaluating other ways to configure a NAT device.

When taking a helicopter view of NAT configuration we are considering the idea of viewing the logical NAT module that we configure as a multi-sided (more than 2 interfaces) device that can handle more then 2 IP realms, see figure 5.2. The ups and downs are how readable and understandable the configuration of the router will be compared against how complex the interpretation of the actual configuration will be.

In figure 5.1 below we show a multi way NAT implemented with 3 2-sided NAT modules and in figure 5.2 we see the same NAT device now implemented as one logical unit. The difference is that we view the first model as consisting of 3 NAT devices — each with an inside and an outside interface and the last model as consisting of one device with 3 interfaces and the XORP NAT module is responsible for generating a matching configuration for each (real) NAT module so the total NAT setup functions as described by the configuration.

The multi way NAT module in figure 5.2 is an integrated box with the same functionality as the model shown in figure 5.1 that consists of 3 NAT modules each providing 2-way NAT functionality.

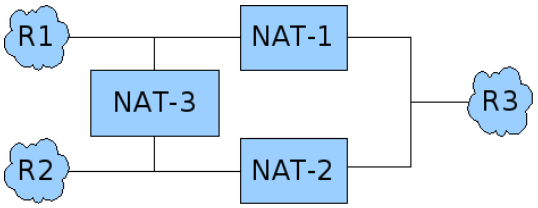


Figure 5.1: 2 way NAT. 3 realms connected with 3 NAT devices.

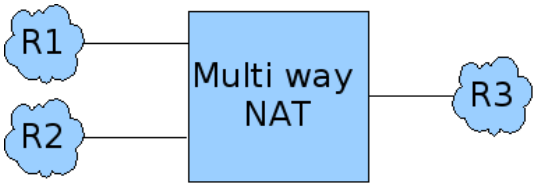


Figure 5.2: Multi way NAT with 3 realms.

5.2 The network environment used in analyzing the model

The following 3 realms: “R1”, “R2” and “R3” are used throughout the examples. The name of the interface where each realm is connected to is also listed. Figure 5.2 and 5.1 shows the two different views we are considering to NAT. Other conditions are listed in the description of each example.

Realm 1: Name: R1, IP-subnet: 192.168.0.0/24, Interface name: em1.1

Realm 2: Name: R2, IP-subnet: 172.16.0.0/18, Interface name: em2.1

Realm 3: Name: R3, IP-subnet: 10.0.0.0/8, Interface name: em3.1

Our goal is to translate the IP hosts from the two local realms R1, and R2 to be able to reach services on the R3 realm, and to allow IP hosts at the R1 to reach servers located at the R2 realm and vice versa.

We have the following NAT options to choose from: 1:1 static NAT translation or dynamic NAT/NAPT translation depending on whether the IP addresses are overlapping or not in the two realms being translated.

1:1 static NAT can be used if the IP-ranges at the two networks are not overlapping. This might seem like a waste of IP-addresses as the hosts each have an IP-address at both networks. For all the traffic that by nature can be translated by dynamic NAT we will prefer this.

The R1 and R2 connectivity to the R3 realm can be solved by using a NAPT translation where R1 and R2 are the inside NAT interface.

The R1 to R2 and R2 to R1 connectivity is a bit harder and can be split up in a number of cases: No overlapping IP-address ranges, overlapping IP-address ranges which we will now describe in detail.

No overlapping IP-address ranges. Realms R1 and R2 are not overlapping IP-ranges, which enables the possibility to use plain IP routing between these instead of NAT. This requires that the R1 IP-address range are available in the R2 realm and vice versa. (e.g. that R1 IP range is unallocated in realm R2) If R1 and R2 belongs to the same realm pure IP routing would be enough to solve the connectivity between them. As we are now in two different realms we need to define the address range either by importing[1] the address space from R1 into R2 and the R2 address range into R1 if the router has this capability or by using 1:1 static NAT between R1 and R2 with the same IP-address range defined at each interface of the NAT module, which in this case will work as translation of IP-addresses. This will change the realms only but not the IP-address of packets translated.

Overlapping IP-address ranges When the R1 and R2 IP-addresses are overlapping address ranges or one of the address ranges are not available in the other realm — then either NAPT with an IP-address for the outside NAT interface in each direction has to be used together with dynamic NAPT to ensure the connectivity alternatively if a 1:1 NAT is wanted, hosts at R1 will have to be translated to another IP range in the R2 realm and vice versa. In practice this is possible, but it creates the problem that a given server is to be contacted at an IP-address depending of what network (R1, R2 or global) the client are located at. Services like DNS would have to be configured to provide a split-view of the DNS world for the specific servers. One view for R2 hosts seen from R1 and another view for R1 hosts seen from R2 which quickly will clutter the whole setup and soon would be in risk of being too complicated for practical uses.

Functions like this would be nice to be able to express in the configuration syntax, but with the rather primitive native NAT modules (not Click) we doubt that these can be used to implement advanced NAT like this.

[1] Importing of IP-addresses from one realm to another is a technique that makes IP-addresses from one realm visible and addressable from another realm. Importing of IP-addresses must be defined for each direction separately.

5.3 Can we configure a multi sided NAT

Seen in this perspective we suggest that the XORP NAT configuration language has definable names for each side of the NAT module and that the configuration language can define realm names for the "inside" and the "outside" NAT interface. This way we will be able to map the NAT modules – despite a 2 or "multi way" NAT model is being used – into a larger network configuration with realm names as the connecting element between possible independent IP subnets. An example configuration language is shown in figure 5.3 at page 43.

The example in figure 5.3 at page 43 consists of 2 main parts. The `nat-realm` definitions (figure 5.3 line 2 – 26) which defines what IP addresses are defined within each realm and the `nat-dynamic` part (figure 5.3 line 28 – 59) which defines the NAT translations. The example we created only contains dynamic NAT definitions.

The 3 realms: "R1", "R2" and "R3" are defined as described in figure 5.3 (lines 02 – 26). Each realm definition consists of a realm-name and a number of `ip-def` clauses which each defines the interface name and IP address ranges that belongs to this realm.

The `nat-dynamic` translation part (figure 5.3 line 28 – 59) defines 3 translation mappings which are named "inside-outside", "r1-r2" and "r2-r1". Each mapping consists of one or more `inside` and `outside` clauses, that connects the defined realms to the dynamic-nat interfaces. As we define dynamic NAT, one side of a dynamic NAT module has the inside interface or the "many" IP-addresses that are translated to the outside interface or the "fever" IP addresses side.

The `map inside-outside` statement (figure 5.3 line 29 – 40) defines a dynamic NAT translation `map` from the two inside realms R1 and R2 to outside realm R3 with global the IP-address at 10.0.0.2.

The dynamic-nat rule named "r1-r2" (figure 5.3 line 41 – 49) defines a NAT translation `map` with inside interface in realm R1 and outside interface in realm R2 with the global IP-address 172.16.0.2.

The dynamic-nat rule named "r2-r1" (figure 5.3 line 50 – 58) defines a NAT translation `map` with inside interface in realm R2 and outside interface in realm R3 with the global IP-address 192.168.0.2.

The example configuration language in figure 5.3 has several new constructs which we will describe briefly:

`nat-realm <realm-name>` defines the realm name and IP-addresses and interfaces of each realm.

`ip-def <ip-def-id>` defines a named node for an IP definition. The node name is stored in the "ip-def-id" and is only used to distinguish between several `ip-def` nodes in the same `nat-realm` node. The `ip-def` node contains one of the following sub-nodes: `ip-address`, `ip-range` or `ip-net`. Each `ip-def` node has a `label` parameter that defines a label to be used in the `inside` or `outside` clauses of the map statements.

`label: <label-name>` The `label` parameter holds the "label-name" of an `ip-def` node. The "label-name" is used in the `map` statements to reference what IP-address definition a `map` definition connects to each interface.

`ip-address: <ip-address>` defines a single IP-address.

`ip-range: <ip-range>` defines a IP-address range. The range consists of a starting IP-address and an ending IP-address.

`ip-net: <ip-net>` defines a IP-network. An IP network consists of the IP-network address and the subnet, that defines the size of the IP-net. Only valid IP-network addresses can be defined.

```
protocols {
    nat {
        nat-realm R1 {
            ip-def net1 {
                description: "Production dept."
                label: R1id
                interface: em0.1
                ip-net: 192.168.0.0 255.255.255.0
            }
        }
        nat-realm R2 {
            ip-def net2 {
                description: "Sales dept."
                label: R2id;
                interface:  em1.1
                ip-net: {172.16.0.0 255.255.192.0 }
            }
        }
        nat-realm R3 {
            ip-def net3 {
                description: "Company backbone"
                label: R3id;
                interface:  em3.1
                ip-net: {10.0.0.0 255.0.0.0 }
            }
        }

        nat-dynamic {
            map inside-outside {
                inside r1 {
                    label: R1id
                }
                inside r2 {
                    label: R2id
                }
                outside r3 {
                    label: R3id
                    ip-address: 10.0.0.2
                }
            }
            map r1-r2 {
                inside r1i {
                    label: R1id
                }
                outside r2o {
                    label: R2id
                    ip-address: 172.16.0.2
                }
            }
            map r2-r1 {
                inside r2i {
                    label: R2id
                }
                outside r2o {
                    label: R1id
                    ip-address: 192.168.0.2
                }
            }
        }
    }
}
```

Figure 5.3: Example of multi way NAT configuration.

`interface: <interface-name>` specifies a name for an network interface of the operating system to be used to bind the defined IP networks to an network interface.

`nat-dynamic` defines a dynamic NAT translation rule which consists of `map` definitions that each has `inside` and `outside` definitions.

`map <map-name>` defines a NAT translation module that has one or more `inside` and `outside` interfaces that is connected to defined IP-addresses selected from the `nat-realm` definitions. Each definition consists of a realm and IP definitions either by direct address definitions or by referring to the `label` from the `nat-realm` statements.

`inside <node-id>` defines what realm (or IP definitions) that are attached to the `inside` interface of the NAT module defined by the `map` statement that the current node is a child node of. The "node-id" name is not used to any configuration functionality, but is only used to distinguish between the various `inside` or `outside` nodes of a `map` statement.

`outside <node-id>` defines the realm that is attached to the `outside` interface of a NAT module defined by the `map` statement that the current node belongs to. If `ip-address` is defined within a `outside` statement, it will be the global IP-address of the outside interface. If the interface already has another IP-address assigned, the IP-address defined in this clause will be an IP-alias-address of the interface. The IP-address defined must not be assigned to any other interfaces or host.

Summary

As we have seen in this example configuration language, it is hard to avoid defining what interface is the "inside" and what is the "outside" interface of a NAT module. There must to be a way to express what side of a NAT module will have the global IP-addresses. So even if the "multi way NAT model" might seem more practical, we still need to express NAT modules as two sided entities - or as n-sided entities where the global side is defined separately, which we believe is almost the same thing.

5.4 What did we learn from the multi way NAT example

We have created and evaluated an example configuration language syntax presented with an example in figure 5.3 at page 43 of a multi way NAT device. As we see in the dynamic-nat translation example (figure 5.3) we can not avoid defining "inside" and "outside" interfaces as the properties of dynamic NAT basically are that one side has the "many" IP addresses and the other side has the "few" IP addresses. Even when we try to describe the 3-way NAT (the "inside-outside map" line 29-40) we need to supply information about the "inside" and the "outside" due to the nature of NAT.

With this argument we will stop considering other than 2-way NAT configuration support, but prepare the configuration language for more NAT translation rules and NAT modules in the same router configuration.

Despite the fact that we really like the idea of defining realms separate as in the `nat-realm` clauses and using these later in the `map` statements, we see that beside this we still need to specify the IP address information for the "outside" interfaces in the `map` clause to tell dynamic NAT what IP address to use for this.

In this perspective, we overall believe that it will be to much configuration to type in when we are defining realms like in the example we have seen, and that the information needed to understand what NAT translations are defined will be spread over too many places

in the configuration file. Therefore we will try to cut the realm part down to a minimum - e.g. just the name and then concentrate at the NAT configuration directly.

5.5 Some details of Load Sharing NAT configuration

When analyzing LS-NAT it seems clear that we need a flexible way to be able to specify IP-addresses together with protocol information.

Protocol information is actually containing two item types, one for the TCP and UDP protocol specification and one for the port specification. In network configurations you often see very detailed specifications for IP-addresses and protocol/port details. The LS-NAT example for the FreeBSD `natd` shows the problem. The natd configuration line:

```
/sbin/natd -redirect_address tcp www1:http,www2:http,www3:http www:http
```

Maps the TCP http port (TCP port 80) of the 3 web servers "www1", "www2" and "www3" to the outside load shared IP address "www" TCP port 80. (In a real configuration the symbolic addresses: "www1", "www2" and "www3" will be exchanged with IP-addresses). To express the LS-NAT line above in our suggested configuration language we need to express multiple IP addresses and matching port numbers in an `inside` or `outside` clause. The example we just saw in the example of multi way NAT in figure 5.3 at page 43 showed multiple IP-addresses but did not include port number information, which we will need to support in the configuration language.

This means that we must find a way to express protocol and port number(s) for an IP definition which we will discuss further in section 5.6 at page 46. We also need to support multiple IP definitions for an `inside` or `outside` clause.

In the examples (figure 5.4 lines 100 – 128 and figure 5.5 lines 200 – 247) we show two examples of the same configuration expressed in two syntactic different configuration languages that we have been evaluating. During both examples "http" means port 80, "www1" has IP-address 192.168.1.5, "www2" has IP-address 192.168.1.8, "www3" has the IP-address 192.168.1.12 and "global-www" has the IP address 80.80.80.80.

In the example in figure 5.4 (line 100 – 128) there are 3 `inside` nodes, and one `outside` node contained in the `map` statement in line 103 – 125. The node structure is much alike the configuration structure for `natd`.

The opposite solution (multiple outside nodes and a single inside node) is not possible in `natd` with load sharing NAT, then the NAT must be turned around.

We have a second configuration example that expresses the same LS-NAT translation in figure 5.5 at page 47 (lines 200 – 247). Each `map` clause has exact one `inside` and one `outside` clause. This requires that the underlaying XORP NAT module detects the 3 equal outside definitions as identical definitions for creating the correct configuration line for `natd`. The `map` configuration lines (figure 5.5 lines 203 – 215) defines the translation `map` statement for the"www1" web server. The lines (217 – 230) defines the `map` statement for the "www2" web server and so forth.

From the two examples of LS-NAT configuration languages in figure 5.4 at page 46 and figure 5.5 at page 47 we see that we need to keep the information together in the (most) compressed form. Here it is shown by having only one `outside` clause for a NAT device with multiple `inside` clauses. This way we avoid the task of re-assembling the information to configure the NAT modules similar to `natd` which need one outside IP-definition and a number of inside IP-definitions as its configuration command. To avoid having to re-assemble this information when generating a configuration for "natd" we choose to have the possibility to express the condensed form (as in figure 5.4) in the configuration language and in the internal data structures of the XORP NAT module.

We also see that IP-addresses, protocol and sub-protocols (TCP/UDP ports) takes up some lines in each definition clause. Here we have used a compressed format in the `protocol:` parameters to save lines in the examples. In XORP today this will have

```
100: protocols {
101:     nat {
102:         ls-nat {
103:             map web-loadshare {
104:                 description "load sharing of web servers"
105:                 outside global {
106:                     realm: outside
107:                     ip-address:80.80.80.80
108:                     protocols: tcp: http
109:                 }
110:                 inside www1 {
111:                     realm: inside-ws
112:                     ip-address: 192.168.1.5
113:                     protocols: tcp: http
114:                 }
115:                 inside www2 {
116:                     realm: inside-ws
117:                     ip-address: 192.168.1.8
118:                     protocols: tcp: http
119:                 }
120:                 inside www3 {
121:                     realm: inside-ws
122:                     ip-address: 192.168.1.12
123:                     protocols: tcp: http
124:                 }
125:             }
126:         }
127:     }
128: }
```

Figure 5.4: Example 1 of LS-NAT configuration example.

to be expressed in two separate lines: one for the protocol and one for the port number. In the next section we define the format used for the `protocol:` parameter in the examples.

5.6 Protocol and sub-protocol specification language

In figure 5.6 we propose a language for writing protocols and port numbers in the configuration file and at the CLI interface. We are going for creating of a generic language for this, which is why we are using the term "sub-protocol" instead of the term "port", as the term "port" is only valid in the TCP and UDP protocols domain. The `protocols:` statement (figure 5.5 at page 47) is used to specify the protocol and sub-protocols in use for an actual translation.

The language we are proposing is created so that more protocols and sub-protocols can be specified at the same text-line after something else – e.g. IP addresses has been specified. When the starting point `protocols-stmt` is used, the beginning keyword "protocols:" is marking the beginning of a protocol and sub-protocol definition at the line. If only a range of protocol and sub-protocols is being specified at a given configuration parameter the "protocols:" keyword is not needed and the `protocols-def` statement can be used as the starting point. If just a range of sub-protocols for the same protocol is to be specified the `sub-protocol-list` starting point can be used.

We will give a few examples of the protocol and port definition language described in figure 5.6. The example uses the `protocols-stmt` starting point:

```
protocols: tcp: 22,33,44-55, udp: 22,33,66-70, 88
```

The example above specifies this set of protocols and sub-protocols: TCP protocol port: 22, 33, 44, 45 ,46 ,47 48, 49, 50, 51, 52, 53, 54, 55 and UDP protocol port: 22, 33, 66, 67, 68, 69, 70 and 88. Protocol and sub-protocol names can be used instead of the numbers, so the following definition has the exact same meaning (ssh is a well known service name for port 22):

```
protocols: tcp: ssh,33,44-55, udp: ssh,33,66-70, 88
```

```
protocols {
    nat {
        ls-nat {
            map web-loadshare-1 {
                description "loadshare for web server www1"
                outside {
                    realm: outside
                    ip-addres:80.80.80.80
                    protocols: tcp: http
                }
                inside {
                    realm: inside-ws
                    ip-address: 192.168.1.5
                    protocols: tcp: http
                }
            }

            map web-loadshare 2 {
                description "loadshare for web server www2"
                outside {
                    realm: outside
                    ip-addres:80.80.80.80
                    protocols: tcp: http
                }

                inside {
                    realm: inside-ws
                    ip-address: 192.168.1.8
                    protocols: tcp: http
                }
            }

            map web-loadshare 3 {
                description "loadshare for web server www3"
                outside {
                    realm: outside
                    ip-addres:80.80.80.80
                    protocols: tcp: http
                }
                inside {
                    realm: inside-ws
                    ip-address: 192.168.1.12
                    protocols: tcp: http
                }
            }
        }
    }
}
```

Figure 5.5: Example 2 of LS-NAT configuration example.

```
protocols-stmt       ::= protocols: <protocols-def>
protocols-def        ::= <protocol>: <sub-protocol-list> [<protocols-def>]

sub-protocol-list    ::= <sub-protocol-spec> [, <sub-protocols-spc>] [, <protocols-def>]...
sub-protocol-spec    ::= <<sub-protocol-no>
                       | <service-name>>
                       | <<sub-protocol-range>
                       | <service-name-range>>

protocol             ::= <protocol-no>
                       | <protocol-name>
protocol-no          ::= <digits>
protocol-name        ::= tcp | udp | icmp | <term from table A.1 or A.2>

sub-protocol-no      ::= <digits>
sub-protocol-range   ::= <sub-protocol-no> - <sub-protocol-no>

service-name         ::= smtp | ftp | ssh | http | <well-known-port-term >
service-name-range   ::= <<service-name> | <sub-protocol-no>>
                         - <<service-name> | <sub-protocol-no>>

well-known-port-term ::= <well-known-services term defined by iana.org >

digits               ::= <digit>...
digit                ::= <0|1|2|3|4|5|6|7|8|9>

letters              ::= <letter>...
letter               ::= <a..z|A..Z>
```

Figure 5.6: Proposed protocol and port definition language.

The protocol and sub-protocol language will be used in the IP specification language that we describe in the next section.

5.7 The IP specification language definition

The XORP configuration language currently have a defined way of specifying IP-addresses and subnet masks at the same configuration line (the ipv4net data type). But XORP does not have a way to define either an IP-address, an IP-subnet or an IP-range at the same configuration line, a separate configuration line and data type for each of these has to be used in the configuration syntax.

Quite some places in the configuration file we will be able to reduce the number of parameters of the configuration clauses (nodes) and at the xrl interface by being able to specify IP addresses and subnets at the same line. This can be done by using a string variable to carry a more complex syntax for these definitions in the configuration file to the xorp NAT module. We have defined a language to define either an IP-address, IP-subnet or an IP-range that is described in figure 5.7 at page 49. We are using the following notations in the definitions: **DDD:** means decimal number from the range 0 .. 255. **SS:** means 2 digit decimal number stating the number of 1-bits in the subnet mask. The valid range is 0 .. 24. **MMM:** means 3 decimal digits of a subnet mask and can only be one of the numbers: {255, 252, 248, 240, 224, 192, 128, 0} with the limitations that usually applies to subnet masks.

The proposed syntax is shown in figure 5.7 below. We only describe IPv4 addresses and subnet masks. IPv6 addresses and subnet masks follows the same idea. The IPv6 syntax is described in the syntax for writing IPv6 IP addresses in figure 5.9 at page 49.

Below is an example of the proposed IP definition language from figure 5.7 The example has 2 lines with the same semantic meaning: IP-subnet address 10.200.18.0 with subnet mask 255.255.255.0.

```
10.200.18.0/24   or  10.200.18.0/255.255.255.0
```

Figure 5.8 below contains the language definition for the IPv4 IP specification language.

```
DDD.DDD.DDD.DDD defines a single IP address.
DDD.DDD.DDD.DDD/SS defines an IP subnet with the subnet mask of ss bits.
DDD.DDD.DDD.DDD/MMM.MMM.MMM.MMM defines an IP subnet with the subnet mask.
DDD.DDD.DDD.DDD .. DDD.DDD.DDD.DDD defines an IP address range.
(alternatively a dash can be used instead of the .. delimiter)
```

Figure 5.7: IPv4 address and subnet mask specification language.

```
IP-statement          ::= <IPvX> [<IP-statement>...]
IPvX                  ::= <IPvX-Addr>  | <IPvX-Net> | <IPvXRange>
IPvX-Addr             ::= <IPv4-Addr>  | <IPv6-Addr>
IPvX-Net              ::= <IPv4-Net>   | <IPv6-Net>
IPvX-Range            ::= <IPv4-Range> | <IPv6-Range>
IPv4-Addr             ::= <IPv4-Addr-part>.<IPv4-Addr-part>
                          .<IPv4-Addr-part>.<IPv4-Addr-part>
IPv4-Net              ::= <IPv4-Addr> | <IPv4-Subnet>
                          |  <IPv4-Addr> | <IPv4-Subnet-mask>
IPv4-Range            ::= <IPv4-Addr> .. <IPv4-Addr>
IPv4-Addr-part        ::= <decimal number in the range of 0-255>
IPv4-subnet           ::= / <decimal number in the range of 0 - 24>
IPv4-Submet-mask      ::= / <IPv4-Subnet-mask-part>.<IPv4-Subnet-mask-part>
                            .<IPv4-Subnet-mask-part>.<IPv4-Subnet-mask-part>
IPv4-Submet-mask-part ::= <decimal number in the range of 0.255>
```

Figure 5.8: IP address and subnet mask language syntax for IPv4 IP addresses.

Figure 5.9 below contains the language definition for IPv6 addresses. IPv6 addresses are not used in the NAT module - but we print it here to complete the language definition.

```
IPv6-Addr      ::= <IPv6-Addr-part>:<IPv6-Addr-part>
                   :<IPv6-Addr-part>:<IPv6-Addr-part>
                   :<IPv6-Addr-part>:<IPv6-Addr-part>
                   :<IPv6-Addr-part>:<IPv6-Addr-part>
                   :<IPv6-Addr-part>:<IPv6-Addr-part>
                   :<IPv6-Addr-part>:<IPv6-Addr-part>
                   :<IPv6-Addr-part>:<IPv6-Addr-part>
                   :<IPv6-Addr-part>:<IPv6-Addr-part>
IPv6-Net       ::= <IPv6-Addr> | <IPv6-Subnet>
IPv6-Range     ::= <IPv6-Addr> .. <IPv6-Addr>
IPv6-Addr-part ::= <hexadecimal number in the range of 0-ff>
IPv6-subnet    ::= / <decimal number in the range of 0 - 128>
```

Figure 5.9: IP address and subnet mask language syntax for IPv6 IP addresses.

A further reduction of the number of parameters used in the configuration language and at the xrl interface for the XORP NAT module, can be achieved by adding the protocol specification from section 5.6 to the same configuration line as the IP address specification defined in figure 5.8 which would further reduce the xrl interface with the `protocols:` parameter (see figure 5.4 at page 46). We only show IPv4 address examples here but the format applies to IPv6 addresses too. The suggested extension is shown in figure 5.10 below.

The example below is an example of the language defined in figure 5.10. It defines TCP port 80 (http) for IP-address 10.200.18.1

```
10.200.18.1:tcp:http
```

Extension of the “IP address and subnet mask language” in figure 5.8 at page 49 to add the suggested protocol extension from section 5.6. (The line below is also part of the definition and is included in figure 5.12 at page 50.):

```
IPvX-protocol-statement   ::= <IPvX>:<protocols-def>
```

We can further extend the language syntax with named interfaces and named virtual interfaces by adding the following lines to the definition in figure 5.10 at page 50. This would

```
DDD.DDD.DDD.DDD:<protocols-def>
DDD.DDD.DDD.DDD/SS:<protocols-def>
DDD.DDD.DDD.DDD/MMM.MMM.MMM.MMM:<protocols-def>
DDD.DDD.DDD.DDD ..  DDD.DDD.DDD.DDD:<protocols-def>
```

Figure 5.10: IP address, subnet mask, protocol and sub-protocol definition language.

combine the parameters for the IP-address with the `interface:` and `vif:` configuration parameters. Creating one configuration parameter that can take the following types of parameters: IP-address, IP-subnet, IP-range, IP subnet mask, protocol, sub-protocol, interface and virtual interface (vif) arguments. Keywords and the syntax identifies each part when parsing the line.

```
interface: <interface-name> vif: <vif-name>:[<protocol-def>]
```

Examples of the syntax above that defines the interface named “em0” with the virtual interface “em0.1” and TCP protocol port 80 (http).

```
interface: em0 vif:em0.1:tcp:80  or  interface: em0 vif:em0.1:tcp:http
```

This can be expressed by adding the extension shown in figure 5.11 to the IP specification language in figure 5.8 at page 49.

```
The IPvX line below redefines the previous definition.

IPvX               ::= <IPvX-Addr> | <IPvX-Net> | <IPvXRange> | <Named-Interface>
Interface          ::= interface: <Interface-name> vif: <Vif-name>
Interface-Protocol ::= interface: <Interface-name> vif: <Vif-name>[:<protocols-def>]
Interface-name     ::= <letter>[<letters>|<digits>]...
Vif-name           ::= <letter>[<letters>|<digits>]...
```

Figure 5.11: Extension of the IP definition language in figure 5.8 and figure 5.9. The IPvX definition here redefines the definition from figure 5.9.

The accumulated result of the design of IP address, protocol, sub-protocol, interface and virtual interface description syntax is listed in figure 5.12. We name the new overall starting point for “IPvX-def” which maps all the earlier definitions together. We name the new overall starting point for “IPvX-def” which maps all the IP address definitions together. If only protocol definitions are wanted in a configuration line, the “protocol-stmt”, “protocols-def” or the “sub-protocol-list” entities must be used directly. This completes the design of the specification languages for XORP NAT configuration.

```
IPvX-def        ::= <IPvX>
                  |  <IPvX>:<protocols-def>
                  |  <Interface>
                  |  <Interface-Protocol>
```

Figure 5.12: Complete language for definition of IP addresses, protocols and interfaces. The definitions in the figures 5.6 at page 48, figure 5.8 at page 49, figure 5.9 at page 49 and figure 5.11 at page 50 are all parts of the full language definition.

A last note about the suggested languages

We have not yet implemented the proposed syntax in the project as we were quite late in the project when we got the ideas for the above IP address and port specification languages. But being able to use this syntax instead of the originally suggested would reduce the configuration language with quite a number of individual configuration lines.

Configuration freeze function

When typing in XORP commands each command will normally be propagated directly to the running configuration. The generation of the actual command for the underlaying NAT module (e.g. Click or natd) will then be executed each time a new configuration node is entered or changed. We suggest having a configurable parameter named `config-hold` in the `nat` clause that freezes the current running NAT configuration until a full configuration is typed in. We consider implementing this feature by "blocking" the generation of new commands to the underlaying NAT system when this parameter is set.

5.8 The full XORP NAT configuration language

This is the complete XORP NAT configuration language. The various parts are described in section 5.9 starting at page 52. This version is the implemented version of the configuration language. The "Final remarks" subsection starting at page 57 lists the optimized version of the configuration that uses the newly defined "IPvX-def" language. All nodes that have an "id" parameter after the node keyword can exist in multiple instances each which must have unique names. This apply to the `map`, `inside`, `outside`, `nat`, `native-nat` and `ip-divert` clauses inside the `fea` clause of the presented language.

```
protocols {
    nat {
        disable: {disabled:bool}
        config-hold: <hold:bool>  // hold-commit function for all nat and
                                  // realm changes.

        static-nat {
                disable: <disabled:bool>

            map <map-id:txt> {
               description: <description:txt>
               disable: <disabled:bool>

               inside <id:txt> {
                   realm: <srcrealm:txt>
                   ip-address: <ip:ipv4>
                   ip-range: <ipfrom:ipv4> - <ipto:ipv4>
                   ipnet: <ipnet:ipv4net>
                   interface: <interface:txt>
                   vif: <vif:txt>
                   protocols: <protocols:txt>
               }
               outside <id:txt> {
                   realm: <destrealm:txt>
                   ip-address: <ip:ipv4>
                   ip-range: <ipfrom:ipv4> - <ipto:ipv4>
                   ipnet: <ipnet:ipv4net>
                   interface: <interface:txt>
                   vif: <vif:txt>
                   protocols: <protocols:txt>
               }
            }
        }

        dynamic-nat {
            disable: <disabled:bool>

            map <map-id:txt> {
                description: <description:txt>
                binding: <binding:txt>
                inside <id:txt> {
                    realm: <srcrealm:txt>
                    ip-address: <ip:ipv4>
                    ip-range: <ipfrom:ipv4> - <ipto:ipv4>
                    ipnet: <ipnet:ipv4net>
                    interface: <interface:txt>
                    vif: <vif:txt>
                    protocols: <protocols:txt>
                }
                outside <id:txt> {
                    realm: <destrealm:txt>
```

```
                    ip-address: <ip:ipv4>
                    ip-range: <ipfrom:ipv4> - <ipto:ipv4>
                    ipnet: <ipnet:ipv4net>
                    interface: <interface:txt>
                    vif: <vif:txt>
                    protocols: <protocols:txt>
                }
            }
        }

        ls-nat {
            disable: <disabled:bool>

            map <map-id:txt> {
                description: <description:txt>
                scheduler-type: <scheduler:txt>
                inside <id:txt> {
                    realm: <srcrealm:txt>
                    ip-address: <ip:ipv4>
                    ip-range: <ipfrom:ipv4> - <ipto:ipv4>
                    ipnet: <ipnet:ipv4net>
                    interface: <interface:txt>
                    vif: <vif:txt>
                    protocols: <protocols:txt>
                }
                outside <id:txt> {
                    realm: <destrealm:txt>
                    ip-address: <ip:ipv4>
                    ip-range: <ipfrom:ipv4> - <ipto:ipv4>
                    ipnet: <ipnet:ipv4net>
                    interface: <interface:txt>
                    vif: <vif:txt>
                    protocols: <protocols:txt>
                }
            }
        }
    }

    fea {
        status-native-nat: <ok:bool>

        nat <id:txt> {
            native-nat <id:txt> {
                disable: <disabled:bool>
                nat-command: <nat-command:txt>
            }

            status-ip-divert: <ok:bool>

            ip-divert <id:txt> {
                disable: <disabled:bool>
                ip-divert-rule: <ip-divert-rule:txt>
            }
        }
    }
}
```

5.9 Description of the XORP NAT configuration language

The XORP NAT module is identified with a `nat` node which is a sub element of the top level (configuration tree wise) `protocols` node.

All definitions regarding the XORP NAT module are located under the `nat` clause in the configuration language except the `click` and the `natd` interface definition which is located under the top level `fea` configuration node.

The XORP configuration is structured as a tree structure. Each node or leaf in the configuration tree has a name, the individual element of the configuration tree is addressed either absolutely or relatively from the current node just like filenames in a UNIX file system are. Each node in the configuration tree can contain named leaves that holds a typed parameter, or named child nodes. If more than one child node with the same name is needed it must have an extra “id” parameter that together with the node name uniquely identifies a given node in the configuration tree structure.

Nodes can contain leaves which can store values and nodes can contain other nodes. This way we can build a configuration tree, which is how the configuration file and the in-memory configuration tree is structured.

The nat clause

The `nat` clause is located under the `protocols` clause (which is not described further here). The `nat` clause is the configuration root for the parameters used in the XORP NAT module. It also defines the location of the *rtrmgr* in-memory configuration tree for the XORP NAT module. The individual sub elements used to configure the XORP NAT module are described in detail in this section.

```
nat {
    disable: <disabled:bool>
    config-hold: <config-hold:bool>

    static-nat {
    }

    dynamic-nat {
    }

    ls-nat {
    }
}
```

The disable parameter

By definition all XORP nodes must have a `disable:` parameter, which when set to `true` will disable the element and everything within it. When a configuration node is disabled the *rtrmgr* act as all configurations within the node (and below) is not configured. This means that all parameters and sub-nodes are reconfigured accordingly by the *rtrmgr*.

This is used to deactivate a configuration element without removing it from the configuration file and the in-memory configuration tree. The `disable:` parameter takes a boolean value that can be either "true" or "false". When the `disable:` parameter is true the node is disabled and when the `disable:` parameter is false the node is active. This description of the `disable:` parameter is valid for all node clauses that has a disable parameter. The `disable:` parameter syntax is shown here:

```
disable: <disabled:bool>
```

The config-hold parameter

The `config-hold:` parameter when set to `true` freezes the running configuration of XORP NAT. This is done by not generating new configurations from the in-memory configuration tree as long as the `config-hold` parameter is true. This allows an operator to type in a new configuration and then commit it to the running system when ready.

```
config-hold: <config-hold:bool>
```

The static-nat clause

The `static-nat` clause holds all the static NAT `map` definitions. Only one `static-nat` clause is allowed in the `nat` clause. More `map` definitions can be held within the `static-nat` clause.

```
static-nat {
    disable: <disabled:bool>

    map <map-id:txt> {
        ...
    }
    ....
}
```

The dynamic-nat clause

The `dynamic-nat` clause holds all the dynamic NAT `map` elements. Only a single `dynamic-nat` clause can be defined in the `map` clause but more `map` clauses can be defined within the `dynamic-nat` clause.

```
dynamic-nat {
    disable: <disabled:bool>

    map <map-id:txt> {
        ...
    }
    ...
}
```

The ls-nat clause

The `ls-nat` clause holds all the Load Sharing NAT `map` elements. As with the `static-nat` and `dynamic-nat` clauses, the `ls-nat` clause can only be defined once within each `nat` clause. Several `map` clauses can be defined within different "map-id" labels.

```
ls-nat {
    disable: <disabled:bool>

    map <map-id:txt> {
        }
        ...
    }
}
```

The map clause

The `map` clause defines the individual mappings of each NAT type. The `map` clause matches a NAT module translation rule with `inside` and `outside` sub nodes.

The map clause has the following syntax:

```
map <map-id:txt> {
    description: <description:txt>
    disable: <disabled:bool>

    scheduler-type:<scheduler:txt> /* lsnat only */
    binding: <binding:txt>         /* dynamic nat only */

    inside <id:txt> {
        ...
    }
    outside <id:txt> {
        ...
    }
}
```

The "map-id" is the node id (or the name) of the `map` node. This is used to identify `map`s from each other – no other function or feature is assigned to the "map-id" parameter.

The `scheduler-type` parameter applies to the `ls-nat` `map` clause only. The parameter is used to define the load sharing algorithm for the mapping. Currently accepted values are `default`, `round-robin` and `least-recent-used`. `Default` selects the default algorithm for the selected NAT module.

The `binding` parameter only applies to dynamic `map` clauses. Binding is used to let the dynamic translation rule prioritize selecting the same (TCP/UDP) ports on both sides of the NAT when creating new dynamic translation entries. This gives protocols such as RPC better chances of surviving the NAT translation. Currently we have only seen this option at the FreeBSD `natd` module. For more information see the description of the FreeBSD natd module option "-same_ports" at page 36.

Valid options are: `default` for normal operation and `static` which sets the "-same_ports" option.

The inside and outside clause

The `inside` and `outside` clause has the same set of child parameters and therefore we describes these together.

The inside clause describes the IP addresses connected to the inside interface of the NAT module, and the outside clause describes the IP addresses connected to the outside interface of the NAT module.

As we are not very happy with this way of specifying the IP definition for a NAT interface we have been working at a more simple interface which will exchange the `ip-address`, `ip-range` and `ipnet` parameters with one common parameter with a well defined syntax. Se the description of the IP specification languages in section 5.7 at page 48 and the description of the protocol specification language in section 5.6 at page 46.

This will probably only change the number of parameters in the configuration file and at the CLI interface, as each type is being identified internally by the XORP NAT module and stored in variables of the relevant types.

Depending of the chosen NAT module, there might be limitations in how complex a configuration that successfully can be converted into a working configuration for the given NAT module can be.

The syntax described here is not intended to limit the possibilities for configuring the NAT functionality, but errors may occur while translating the configuration to a configuration for a given NAT module.

```
inside <id:txt> {
    realm: <srcrealm:txt>
    ip-address: <ip:ipv4>
    ip-range: <ipfrom:ipv4range>
    ipnet: <ipnet:ipv4net>
    interface: <interface:txt>
    vif: <vif:txt>
    protocols: <protocols:txt>
}

outside <id:txt> {
    realm: <destrealm:txt>
    ip-address: <ip:ipv4>
    ip-range: <ipfrom:ipv4range>
    ipnet: <ipnet:ipv4net>
    interface: <interface:txt>
    vif: <vif:txt>
    protocols: <protocols:txt>
}
```

The realm: parameter

The `realm:` parameter is used to name the realm connected to the NAT interface being configured. The realm name is not directly used to name real existing realms, but it is used internally to identify which network are in the same realm. We consider to use the standard names “inside” and “outside” as default assigned values until realms are really supported. Default values for NAT module parameters can be defined in the template file for the nat module. The realm name parameter is a text string.

```
realm: <realm:txt>
```

The ip-address: parameter

The `ip-address:` parameter is used to specify one IP version 4 IP-address for the NAT interface being configured. The syntax used is the usual for IP-addreses e.g: 10.200.18.1

```
ip-address: <ip-address:ipv4>
```

The ip-range: parameter

The `ip-range:` parameter is used to define one IP version 4 IP-range for the NAT interface being configured. The syntax used is the xorp standard with 2 IP-addresses separated with double dots e.g: `10.200.18.1 .. 10.200.18.10` which specifies the IP-addresses from 10.200.18.1 to 20.100.28.10 both addresses given as the parameters are included in the defined range.

```
ip-range: <ip-range:ipv4range>
```

The ipnet: parameter

The `ipnet:` parameter is used to define one IP version 4 IP-subnet for the NAT interface. The syntax used is the standard xorp syntax: `10.200.18.0/24` or `10.200.18.0 255.255.255.0`

```
ipnet: <ipnet:ipv4net>
```

The interface parameter

The `interface:` is a text string that identify an interface by name. This is equivalent to the XORP interface type.

The Interface and virtual interface (vif) - 2 level of nodes configuration model is mandatory to use when defining interfaces in the XORP environment. The `interface:` parameter takes a text string argument with the name of an existing interface at the hosting-OS e.g: `em0`

```
interface: <interface:txt>
```

The vif parameter

Vif means virtual interface. The `vif:` parameter is a string parameter which is used to name an virtual interface. Vif interfaces are sub interfaces of a XORP interface declared by the `interface:` parameter. The `vif:` parameter must match existing named vif interfaces defined in the hosting-OS system e.g.: `em0.1` which is the way binding of the host-OS interfaces to XORP interfaces is done.

```
vif: <vif:txt>
```

The protocols parameter

The `protocols:` parameter (in an `inside` or an `outside` clause) is used to specify the protocols that is affected by the IP specification. If the IP specification covers more IP addresses the protocols specification applies to each IP address in the IP definition.

The parameter type is text and the syntax of the field input, is the syntax defined as the "protocols-def" syntax in section 5.6 at page 46.

```
protocols: <protocol-def:txt>
```

The FEA configuration part

The following configuration elements is the FEA configuration elements that controls which NAT module at the Host OS is performing the actual address translation. We describe each part in detail during the next pages.

The fea clause

The `fea` clause is the top level configuration node for the FEA module. We have added the following set of configuration elements to the already existing set of configuration elements.

The status-native-nat command

The `status-native-nat:` command returns "true" if the native NAT subsystem is enabled in the kernel and configured. If the command returns "false" NAT is not supported (or configured) at the hosting OS.

The fea nat clause

The `nat <id:txt>` clause is the FEA control element for an active NAT element (e.g. natd). The configuration language support simultaneous running NAT instances to support advanced configurations.

The native-nat clause

The `native-nat <id:txt>` clause is the FEA control element for selecting the active NAT element. At FreeBSD this will be "natd". The "id" label holds the natd command name and path (e.g. "/sbin/natd").

The nat-command parameter

The `nat-command:` holds the command that is sent to the NAT device identified by the parent node-id. It is possible to issue a direct command to the NAT subsystem, or see what command is generated by the XORP NAT module and sent to the NAT sub system.

The status-ip-divert command

The `status-ip-divert:` is a check for if IP-DIVERT kernel support is enabled in kernel of the host OS. "True" return value means that the IP-DIVERT subsystem is enabled. A "false" return value indicates that the IP-DIVERT subsystem is not enabled.

The ip-divert clause

The `ip-divert` node identifies a single IP-DIVERT statement. The "id" parameter is used only to distinguish more `ip-divert` nodes from each other.

The ip-divert-rule parameter

The `ip-divert-rule:` holds the IP-DIVERT rule of the node. The rule is inserted "as-is" with the `/sbin/ipfw` command (or similar commands for non FreeBSD hosted systems). See the FreeBSD manual pages for ipfw(8) and divert(4).

Summary

We have defined and described a XORP NAT configuration language that is capable of expressing the desired NAT configurations. We had not yet implemented all the suggested improvements for specifying IP addresses and protocol and port ranges that we suggested in section 5.7 page 48 and section 5.6 page 46.

If the suggested improvements are implemented the `inside` and `outside` configuration clause will change from the form described above to something like the form we

present in figure 5.13 below. We have reduced the number of parameters per NAT interface from 7 to 2. We believe that this form is fulfilling our requirements of being understandable, easy to read and not too complicated to write.

As an example of the reduced IP and protocol specification language suggested in figure 5.12 page 50 we can reduce the inside and outside configuration nodes to only 2 parameters each. To show this we have rewritten the old format configuration example for the LS-NAT setup from figure 5.4 page 46 (lines 105 – 124), to the improved configuration format. The result is presented in figure 5.14 below. The result of this is that 20 configuration lines in the old format now can be written in only 16 lines.

```
inside <id:txt> {
    realm: <srcrealm:txt>
    ip: <ip-and-protocol-def:txt>
}

outside <id:txt> {
    realm: <destrealm:txt>
    ip: <ip-and-protocol-def:txt>
}
```

Figure 5.13: The `inside` and `outside` clauses with the new IP address and protocol specification language used.

```
inside www1 {
    realm: inside-ws
    ip: 192.168.1.5:tcp:http
}
inside www2 {
    realm: inside-ws
    ip: 192.168.1.8:tcp:http
}
inside www3 {
    realm: inside-ws
    ip: 192.168.1.12:tcp:http
}
outside global {
    realm: outside
    ip: 80.80.80.80:tcp:http
}
```

Figure 5.14: A configuration example of the language described in figure 5.13. The example is expressing the same as the lines 105 – 124 in the example presented in figure 5.4 page 46.

5.10 Evaluating the XORP NAT configuration language

We have defined a configuration language for the XORP NAT module which we will evaluate by showing some configurations expressed in the language. We evaluate two examples of static NAT, one example of dynamic NAT and one example of Load Sharing NAT (LS-NAT). Together with the examples we will print the command that is generated to configure the FreeBSD `natd` module.

The most important for our evaluation is to verify that the configuration language is able to express the constructs we have identified during our analysis of NAT implementations and the readability of the configuration.

Static NAPT from IP address and port range

The configuration shown in figure 5.15 below sets up a static NAT translation rule that translates to and from the inside realm located at IP-address 172.17.16.15 TCP port 2300 to 2399 to the outside realm at IP address 80.10.10.10 TCP port 3300 to 3399. This is a

translation that both translates IP addresses and port numbers. When traffic hits TCP port 2301 at the inside IP address it is translated to the outside IP address TCP port 3301 and vice versa.

The global address 80.10.10.10 is configured at interface: em0 vif: em0 which is done outside the nat configuration section of the XORP configuration language.

```
protocols {
    nat
    {
        static-nat {
            map id {
                description: "The outside interface"
                outside outside-if {
                    realm: outside
                    interface: em0
                    vif: em0
                    protocols: tcp: 3300-3399
                }
                inside inside-if {
                    realm: inside
                    ip: 172.17.16.15
                    protocols: tcp 2300-2399
                }
            }
        }
    }
}
```

Figure 5.15: Static NAT translation of outside address of interface em0 vif: em0 TCP port 3300-3399 to inside address 172.17.16.15 TCP port 2300-2399.

The XORP configuration in figure 5.15 matches the natd configuration depicted below.

```
natd -interface em0 -rdirect_port tcp 172.17.16.15/2300-2399 3300-3300
```

Static NAT telnet to local server

The second static NAT example shown in figure 5.16 below is translating an outside NAT address: 80.10.10.10 TCP port 6666 to an inside host at IP address 172.17.16.15 TCP port 23 (telnet). The outside IP address of the host is 80.10.10.10 and is configured at interface: em0 vif: em0 as an alias IP address at the em0 interface. This is done outside of the nat configuration part of the XORP configuration language.

Alias IP addresses at the outside interface is always selected by default when an IP address is defined. To use the native IP address of an interface, the interface: and vif: parameters must be used to specify the interface.

The XORP configuration in figure 5.16 is generating the natd command line below.

```
/sbin/natd -alias_address 80.10.10.10 -redirect_port TCP 172.17.16.15/telnet 6666
```

Load Sharing NAT

The XORP NAT configuration for load sharing a global IP address between 3 internal servers is shown in figure 5.4 at page 46. The example generates the following natd command line:

```
/sbin/natd tcp -redirect_address 192.168.1.5:80,192.168.1.8:80,192.168.12:80
<new-line-inserted> 80.80.80.80:80
```

Dynamic translation example

Figure 5.17 shows a dynamic NAT translation configuration which translates from the inside IP-network 172.17.16/24 to the outside IP-address of interface: em0 vif: em0 which is 80.10.10.1 and allocated via DHCP when the host was booted.

```
protocols {
    nat {
        static-nat {
            map map1 {
                description: "Static nat for access to inside telnet hosts"
                disable: false

                outside out {
                    realm: outside
                    ip: 80.10.10.10
                    protocols: tcp: 6666
                }
                inside ins {
                    realm: inside
                    ip: 172.17.16.15
                    protocols: tcp: telnet
                }
            }
        }
    }
}
```

Figure 5.16: Static NAT translation of 80.10.10.10 port TCP 6666 to inside IP address 172.17.16.15 TCP port 23 (telnet).

The configuration described can be expressed in XORP NAT configuration language like this:

```
protocols {
    nat {
        dynamic-nat {
            map labnet-to-outside {
                description: "lab network"

                outside global-if {
                    interface: em0
                    vif: em0
                }
                inside labnet {
                    ip: 172.17.16.0/24
                }
            }
        }
    }
}
```

Figure 5.17: Dynamic NAT translation rule that translates from inside IP subnet 172.17.16.0/24 to the outside IP address of interface: `em0` vif: `em0`.

From the analysis of the `natd` module in section 4.4 at page 35 we know that the native `natd` module can not specify inside addresses at its command line when configuring dynamic NAT. It has only support to map all known addresses at the host to the global NAT interface. If the address range has to be limited the IPFW divert rule must be used to select the packets that is sent to the `natd` program.

We are solving this by defaulting to not configuring the inside nets at the `natd` command and consider to implement functionality to direct the inside IP address definitions to the IPFW divert rule at a later version.

With these comments in mind, the XORP NAT configuration in figure 5.17 is implemented with the `natd` command below:

```
natd -interface em0 -dynamic
```

Summary

We have evaluated the proposed configuration language and feel confident that the designed language is able to express a wide variety of NAT configurations.

We also believe we have created a configuration language that is readable and understandable despite its rather complicated technical capabilities. One of the thing we would like to comment at is that it is still necessary for an administrator to have a good understanding of the IP configuration of the network interfaces (which is located at another part of the XORP configuration language) to configure or understand a working NAT setup.

Chapter 6

Implementation

Overview

In this chapter we will describe the implementation of the XORP NAT module. First we will describe the environment the module will exist in, then we will describe the XORP NAT module itself and do a breakdown of the module into sub functions, finally we describe each identified sub function in detail and explain the objects related to each sub function.

6.1 The XORP NAT module environment

The module environment in XORP consist of a number of items: The XRL interface, a common set of data types, a process environment and a process state definition. Each XORP module is a self-contained UNIX process that is controlled from the *rtrmgr* process.

The flow of command and communication in XORP is shown in figure 6.1. The figure shows that the *rtrmgr* receives configuration commands from the user interface *xorpsh* The *rtrmgr* builds its own version of the configuration in the in-memory configuration tree. When nodes in the *rtrmgr* internal tree are added, changed or deleted the changes are reflected to the running system by the *rtrmgr* which sends xrl commands to the various XORP modules.

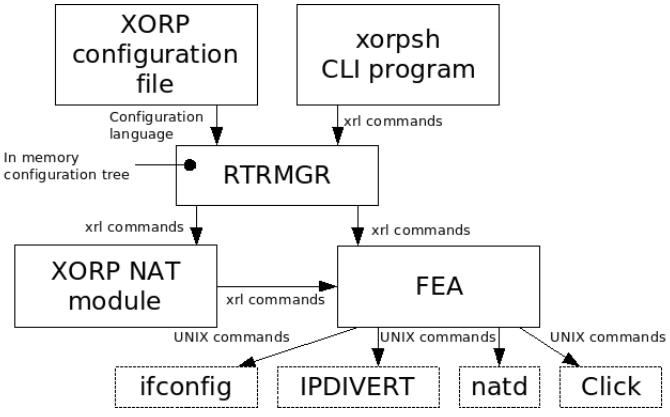


Figure 6.1: Configuration flow within the XORP system. We see XRL commands are sent to the XORP NAT module and to the *fea* which manage the underlaying operating system features by sending UNIX commands to the individual devices or programs.

Figure 6.1 depicts the flow of configuration data from the XORP configuration file and the *xorpsh* via the *rtrmgr* to the rest of the XORP system. The *rtrmgr* sends commands to

the XORP NAT module and sends commands to the non XORP modules *IPDIVERT*, *natd* and the *Click modular router* subsystem via the *fea* module.

6.2 The XORP NAT module

The XORP nat module has a number of internal functions that we will identify and describe to give an impression of the structure of the module. From the helicopter view the XORP NAT module is divided up into tree main functions, the first is the XRL interface part which handles the XRL interface to the rest of the XORP system. The second part is the configuration storage that stores the configuration data received from the *rtrmgr* and the third is the part that reads the content of the storage and generates a configuration for the nat module (Click or natd) that is currently configured. The block diagram of the XORP NAT module is depicted in figure 6.2 and a short description of each sub function is given in the paragraphs below.

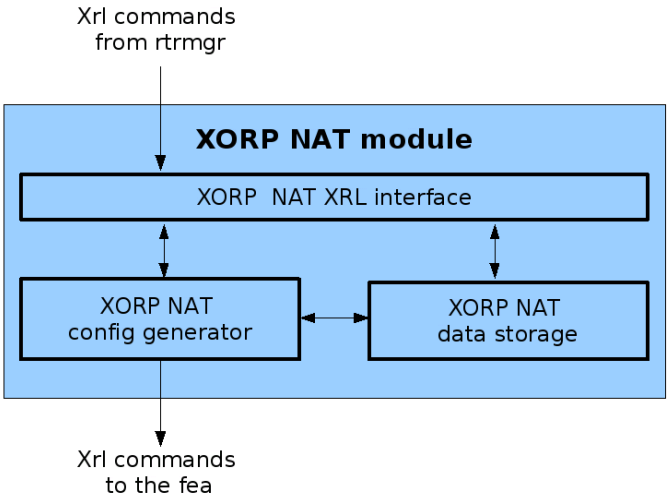


Figure 6.2: Block view of the XORP NAT module. XRL commands is received from the *rtrmgr* via the XRL interface and the configuration generator part sends XRL configuration commands to the *fea* that directs them to the NAT modules at the operating system.

XRL Interface

The XORP NAT XRL interface is where all commands to the module is arriving. The XRL interface has a function for each XRL call that implements the functionality in the module, that it is responsible for.

The Configuration storage

The XORP NAT configuration storage consists of a number of C++ classes that stores the configuration from the *rtrmgr* in a form that makes it usable for generation of nat configuration commands. By this we mean that the modules is storing the supplied informations in a structured way from which it is possible to retrieve the data in a suitable condensed form. Examples of this is if a IP subnet definition is sent to the storage it can be retrieved again as an IP subnet definition or if a range of neighbor sub-protocols (ports) is stored as individual sub-protocols it can be retrieved as one unbroken range. The storage is optimized this way to be able to provide data in a form expected (or usable) by many NAT subsystems.

The NAT configuration generator

The NAT configuration generator reads the stored configuration information and translates these into configuration commands for the supported NAT modules. Each NAT function is represented by a NatMap object and the configuration generator module tries to fit as many NatMap nodes together (e.g. static, dynamic and Load Sharing) to a valid nat configuration. If not all the configuration nodes can be implemented by a given NAT module an error message is emitted.

Other functions

A few other functions exists in the module - among these are: translator functions and the startup and shutdown functions for the module. The translator functions translates between protocol and SubProtocol numeric id to the text (name) representation of the same, and the startup and shutdown functions manage the startup and shutdown of the module logic, XORP logging functions and other internal xorp management functions.

6.3 The XORP NAT XRL interface

The XRL interface part of the XORP NAT module is responsible for handling of the XRL communication to and from the module. The code for the low level part of the XRL interface is generated automatically as a part of the XORP XRL system. We have to write the implementation for each interface to make it perform the desired functions. For more information about the low level part of the XORP XRL system see the "An Introduction to Writing a XORP Process" document [15].

Each time the *rtrmgr* sends a xrl command to the XORP NAT module the module either stores the received configuration data or execute the command related to the xrl command.

The XRL interface for the XORP NAT module has a set of XRL function calls each with a number of parameters associated. We have to provide the implementations for these to perform the desired module functions when receiving a call to the corresponding XRL interface.

Figure 6.3 shows the defined XRL interface for the XORP NAT module

```
set_nat_disable          | Overall module management
get_nat_disable          |
nat_config_hold          |

create_nat_static_map4   | Static NAT interface
delete_nat_static_map4   |
get_nat_static_map4      |

create_nat_dynamic_map4  | Dynamic NAT interface
delete_nat_dynamic_map4  |
get_nat_dynamic_map4     |

create_lsnat_map4        | LS-NAT interface
delete_lsnat_map4        |
get_lsnat_map4           |
```

Figure 6.3: XRL interface functions for the XORP NAT module.

The XRL interface consists of the functions listed in figure 6.3. The functions are described briefly here. A more detailed description of the functions is provided in template file for the XORP NAT module named `nat.xif`.

Below we have a description of all the classes that is used in the XORP NAT configuration storage sub module:

set_nat_disable: This command takes a single boolean argument which when true disables the nat module.

get_nat_disable: Returns the status of the XORP NAT module. The module is in disabled state when the call returns a "true" value.

nat_config_hold: When sent a "true" value, the XORP NAT configuration is "held" (frozen). This means that no new nat configuration command is emitted to the NAT devices (natd or Click) each time the configuration storage changes its content. This way a network administrator can freeze the running configuration, change the configuration and then release the complete new configuration when ready.

create_nat_static_map4: This command creates a static NAT entry which is stored in the configuration storage module. A new NAT command is generated and emitted to the running system depending of the `nat_config_hold` state.

delete_nat_static_map4: This command deletes a static nat configuration. The "id" parameter selects the NatMap node to be deleted. A new NAT command is generated and emitted to the running system depending of the `nat_config_hold` state.

get_nat_static_map4: The command returns the content of the NatMap node that matches the submitted "id" argument.

create_nat_dynamic_map4: Handles the creation of a dynmaic NAT translation rule. There are also a `delete_nat_dynamic_map4` and a `get_nat_dynamic_map4` function call to the dynamic NAT interface. The 3 "dynamic" XRL functions matches the 3 "static" functions with the one exception that there is an `binding` parameter which is used to specify the dynamic "-same_port" option of the natd module.

create_nat_lsnat_map4: Handles the creation of a Load Sharing NAT translation rule. There are also a `delete_nat_lsnat_map4` and a `get_nat_lsnat_map4` function call to the load sharing NAT interface. The 3 "load sharing" XRL functions also matches the 3 "static" functions with the one exception that there is an `scheduler` parameter which is used to specify the scheduler algorithm for the load sharing between internal hosts.

At quite a late state in the project we have envisioned that the 3 NAT interface types (static, dynamic and LS-NAT) has a very similar structure which is also the case when looking at the internal representation of the configuration. So in the next version we will reduce the 3 groups of nat XRL calls to only one group that takes the NAT type as a parameter. The reason to the 3 different interface groups is that we initially believed that the syntactic descriptions of the 3 NAT types would differ more than they ended up doing.

Implementation of the XRL interface

The XORP NAT XRL interface code consists of the xrl function stub-code generated by the XORP XRL system and our code that perform the actual "module relevant" functions.

The auto generated xrl interface stub-code is part of a class hierarchy the is defined by XORP. The object hierarchy is described in the next subsection.

The xrl interface code interacts with the XORP NAT configuration storage, with the FEA and with the XORP NAT configuration generator.

XRL interface Object hierarchy

There is not a large and complex object hierarchy in the XORP NAT module, the hierarchy depicted in figure 6.4 is due to the interface to the XRL interface system. We are responsible for the code in the 2 classes named `XrlNatNode` and `NatNode`. The remaining classes are internal XORP classes that we are using because we communicate with various parts of the *rtrmgr*. The `IfmgrHintObserver` class is responsible for signaling the

NAT module when any interfaces are changed at the *FEA* module. The `ServiceBase` is the base class for XORP services, the `ServiceChangeObserverBase` class is used if synchronous service status change is signaled from the XORP services system (e.g. when a XORP module changes its service state). The `XrlStdRouter` class is the "standard" XRL transmission and reception point of an XRL connection. Finally the `XrlNatTargetBase` class contains the XRL targets for the XORP NAT module as pure virtual functions, which we implement the "real" versions of in the `XrlNatNode` class.

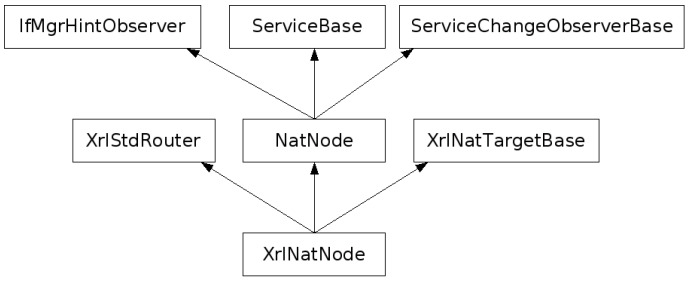


Figure 6.4: Object inheritance for the XrlNatNode class.

The XRL interface classes

The xrl interface classes consists of the following C++ classes:

XrlNatNode class: The XrlNatNode class represents the XRL interface to the XORP system. This is the class where the auto generated code and our own code meets. The function skeletons are automatic generated as pure virtual functions, which we provide the real implementations for.

Each XRL function call has a matching function in this class which is where our code receive the supplied parameters (arguments of the xrl calls) and returns the results back to the calling module via the corresponding set of parameters. XRL parameters are either input or output direction, they can not carry information both ways during an XRL call.

Each XrlNatNode has an associated NatNode object that maintains the XORP state and state changes by the module.

To get an impression of the class and the cooperation with the XRL interface, we list the member functions that exists in the XrlNatNode class in figure 6.5 below.

NatNode class: The NatNode class is responsible for the process state of the module. It performs the state changes from one process state to another according to the XORP process state specifications. The module is most relevant during startup and shutdown synchronization with the rest of the XORP system. We are not describing this class in deeper details at its function is simple to understand when reading the source code and the XORP process state transitions description.

6.4 The XORP NAT Configuration storage

The XORP NAT configuration storage consists of a number of C++ classes that each handles a particular type of configuration data sent to the module. One part handles the defined realms, another part the IP definitions, a third part handles protocols and a fourth part handles the sub-protocols. Other classes handles NAT types (static, dynamic etc.), the IP addresses and realms defined at each NAT interface.

```
XORP NAT XRL commands (Related to NAT functions)
  XrlCmdError XrlNatNode::nat_0_1_set_nat_disable(...);
  XrlCmdError XrlNatNode::nat_0_1_get_nat_disable(...);
  XrlCmdError XrlNatNode::nat_0_1_nat_config_hold(...);
  XrlCmdError XrlNatNode::nat_0_1_create_nat_static_map4(...);
  XrlCmdError XrlNatNode::nat_0_1_delete_nat_static_map4(...);
  XrlCmdError XrlNatNode::nat_0_1_get_nat_static_map4(...);
  XrlCmdError XrlNatNode::nat_0_1_create_nat_dynamic_map4(...);
  XrlCmdError XrlNatNode::nat_0_1_delete_nat_dynamic_map4(...);
  XrlCmdError XrlNatNode::nat_0_1_get_nat_dynamic_map4(...);
  XrlCmdError XrlNatNode::nat_0_1_create_lsnat_map4(...);
  XrlCmdError XrlNatNode::nat_0_1_delete_lsnat_map4(...);
  XrlCmdError XrlNatNode::nat_0_1_get_lsnat_map4(...);

XORP module control interface (Related to the XRL subsystem).
  XrlCmdError XrlNatNode::common_0_1_get_target_name(...);
  XrlCmdError XrlNatNode::common_0_1_get_version(...);
  XrlCmdError XrlNatNode::common_0_1_get_status(...);
  XrlCmdError XrlNatNode::common_0_1_shutdown()
  XrlCmdError XrlNatNode::finder_event_observer_0_1_xrl_target_death(...);
  void XrlNatNode::finder_register_interest_fea_cb(...);
  void XrlNatNode::finder_deregister_interest_fea_cb(...);
  void XrlNatNode::fea_register_shutdown()
```

Figure 6.5: The C++ member functions that exists in the XRL interface implementation of the XORP NAT module.

Implementation of the configuration storage

In the following we will take a short detailed tour through the implementation of the configuration storage object hierarchy and describe the implementation from an bird view with the necessary details needed to get a good understanding of the implementation. All operations of the configuration storage is done via the functions in the XRL interface or from the configuration generator sub module which reads the stored content. We start with describing each object class. None of the classes used for the configuration storage module has inheritance amongst each other.

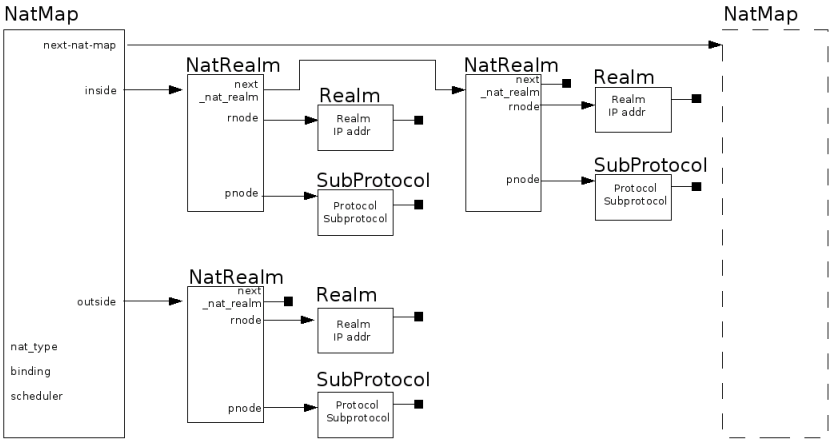


Figure 6.6: Relations between objects in the XORP NAT configuration storage. The black squared boxes symbolizes NULL pointers.

Configuration storage object classes

The following is a description of each object class used to implement the XORP NAT storage module. First we will describe the general issues that are valid for all objects following a detailed description of each object class.

All object classes are designed with the same overall concept in mind. All classes (except the NatNode class) is able to maintain its objects as single linked lists. Each list has an separate defined pointer that holds the start of the linked lists. This pointer must be supplied to all member functions that handles list operations. Methods that only change internal variables of an object does not require the head pointer.

All classes shares a set of defined method names that has the same functionality to the object or the list it is a part of. The following is the list of these ”well known” method names used in the configuration storage C++ classes that we describe later:

count(): Counts the number of elements in the list that matches the supplied parameters. Several versions of `count()` exists to count different parameters.

get(): returns the n-th list member matching the supplied parameters. Implementations exists to match on various parameters.

drop(): Deletes matching nodes from the list. All objects linked from the deleted object are properly destroyed.

insert(): Inserts the “this” object into the list pointed out by the list-header. The element is inserted according to the rules for the actual object type, and all internal pointers are properly adjusted.

remove(): Removes the current “this” object from the supplied list-header. The parent object is located and pointers are adjusted to reflect the removal of the object. The removed “this” object is destroyed.

find_parent_node(): This function searches the list pointed out by the supplied list-header and return a pointer to the parent object to the invoking object. This method is often defined as a private method of the object class.

“internal attributes”: Most internal attributes are accessible by their name without the initial “_” character and can be set and read, by using the proper form of the name. E.g. if the internal variable name is: “_vif” it is read by using this form: “v = vif()” and the variable receives a new value by this form: “vif(”new value”)”.

The object classes we are using in the XORP NAT module configuration store are described below:

NatMap class: The NatMap class keeps track of which NatRealm objects list is mapped to each (inside or outside) interface of a NAT module and the parameters connected to the nat module e.g. the type of nat (static, dynamic etc.) the object represents.

NatMap objects are linked together to a list of all defined NAT functions in the configuration file. The NatMap class contains the NAT type, binding (only used for dynamic NAT) and the scheduler (only used for LS-NAT) information for the NAT translation rule defined by the NatMap node.

NatRealm class: Combines the IP address with protocol and sub-protocol information into one node. Each NatRealm object points to an Realm object with an IP address definition and to a Protocol object with the protocol and sub-protocol definitions that apply to the Realm object.

Each NatRealm object is only linking to exact one Realm object and to one linked list of SubProtocol objects. The NatRealm objects can be linked after each other. This is to be able to have more than one inside or outside IP-definition for a NatMap module.

Realms class: Keeps records of defined realms and holds a pointer to the start of each linked list of class Realm objects. Realm names are identified by a text string.

Realm class: Keeps records of the IP address definitions for a single named realm. The definitions is kept as a linked lists of "class Realm objects" each holding one of the following defined objects: IP address, IP subnet, IP address range, interface name, vif name or IP alias address.

Protocol class: The Protocol class is a linked list of nodes each representing a defined protocol identified by its "protocol number" (e.g. TCP has protocol number 6). Each node holds the pointer to the start of the linked list of "class SubProtocol" objects.

SubProtocol class: The SubProtocol class keeps the records of the defined sub-protocols for a given protocol type. The linked list of SubProtocol objects is consisting of objects with the same protocol-id.

The SubProtocol class always stores the supplied sub protocol information in the most condensed form which means that if sub-protocol 20 and 22 is defined then when the sub-protocol 21 is being defined the class collapse these 3 to a range containing the sub-protocols 20 – 22. The opposite happens when deletion within ranges occurs.

With this we concludes the description of the XORP NAT configuration storage.

6.5 The XORP NAT Configuration generator

The XORP NAT Configuration generator traverses the NatMap object list and maps the NAT translations defined in each of the object. The NatMap linked object list is implemented one node at a time. The first node is converted to a configuration command for the active NAT module at the hosts IP subsystem. The following NatMap objects in the list is adding to the configuration if the content of the nodes is possible to "add" to the configuration for the former objects implemented. If some nodes can not be implemented an error message identifying the failed objects is issued and the administrator has to configure a setup that is possible for the given NAT subsystem.

We did not implement this part of the XORP NAT module and we do not have further details to document here.

6.6 Other functions

Besides the functions mentioned we had created 4 small helper functions that translates protocol names to protocol numbers and sub protocol names to sub protocol numbers and vice versa. The functions which names ends with "t2n" (for text to number) translates from text string to numeric form and the 2 functions which names names ends with "n2t" (for number to text) takes a numeric argument and translates to corresponding text string. The functions are able to support tables with all protocol and sub-protocol names and numbers and is expandable just by adding more table entries to existing tables or new tables to the module. The translations is done by the help of tables that we list in appendix A at page 73. The 4 functions are named like this:

protocolt2n(): The protocolt2n function takes a string argument of a protocol name and translates it to to the corresponding protocol number.

protocoln2t(): The protocoln2t function takes a numeric argument and translates it to the corresponding text string version of the protocol name.

sub_protocolt2n(): The sub_protocolt2n function takes a protocol numeric value and a sub_protocol text string, and converts it to the corresponding sub-protocol numeric value. We have provided tables for TCP and UDP sub protocols.

sub_protocoln2t(): The sub_protocoln2t function takes a numeric protocol value and a numeric sub protocol value and returns a string with the text of the corresponding sub protocol name.

We have added a compile time option that enables the selection between 2 different tables that differs only by their sizes. The large table has all the protocols [36] and ports [37] defined by Internet Assigned Numbers Authority (IANA) at www.iana.org.

6.7 Final remarks

The XRL interface nat_type (static, dynamic or LS-NAT) should be changed to one generic set of XRL interfaces instead of the implemented solution. By doing this we can save 6 XRL functions. We learned this at a rather late time during the project and did not implement these changes in our first version of the code.

We also did not finish implementing the NAT command generating module (for natd or Click commands) due to lack of time.

Besides the comments above we have implemented all the described functionality and support for the XORP logging system in the NAT module.

XORP log is the XORP system log message function that provides a framework for global log files or syslog support from xorp even when the XORP system is build on a modular concept. A small description of using the XORP log functionality can be found in the “An Introduction to Writing a XORP Process” [15].

Appendix A

Names of protocols and sub-protocols (ports)

The XORP NAT has built-in recognition of protocol and TCP/UDP port names. By defining the preprocessor symbol "__NAT_PROTOCOL_T2N_LONG_LISTS__". The long version of the tables with names of protocol and ports (services) are compiled into the module. If the symbol is undefined the short version is used instead.

Protocol names

The short version of the protocol name table is listed in figure A.2 and the long version is listed in figure A.1.

The long version of the supported protocol name and number table:

Decimal	Keyword	Protocol	References
0	hopopt	IPv6 Hop-by-Hop Option	[RFC1883]
1	icmp	Internet Control Message	[RFC792]
2	igmp	Internet Group Management	[RFC1112]
3	ggp	Gateway-to-Gateway	[RFC823]
4	ip	IP in IP (encapsulation)	[RFC2003]
5	st	Stream	[RFC1190,RFC1819]
6	tcp	Transmission Control	[RFC793]
7	cbt	CBT	[Ballardie]
8	egp	Exterior Gateway Protocol	[RFC888,DLM1]
9	igp	any private interior gateway (used by Cisco for their IGRP)	[IANA]
10	bbn-rcc-MON	BBN RCC Monitoring	[SGC]
11	nvp-ii	Network Voice Protocol	[RFC741,SC3]
12	pup	PUP	[PUP,XEROX]
13	argus	ARGUS	[RWS4]
14	emcon	EMCON	[BN7]
15	xnet	Cross Net Debugger	[IEN158,JFH2]
16	chaos	Chaos	[NC3]
17	udp	User Datagram	[RFC768,JBP]
18	mux	Multiplexing	[IEN90,JBP]
19	dcn-MEAS	DCN Measurement Subsystems	[DLM1]
20	hmp	Host Monitoring	[RFC869,RH6]
21	prm	Packet Radio Measurement	[ZSU]
22	xns-IDP	XEROX NS IDP	

Decimal	Keyword	Protocol	References
23	trunk-1	Trunk-1	[BWB6]
24	trunk-2	Trunk-2	[BWB6]
25	leaf-1	Leaf-1	[BWB6]
26	leaf-2	Leaf-2	[BWB6]
27	rdp	Reliable Data Protocol	[RFC908,RH6]
28	irtp	Internet Reliable Transaction	[RFC938,TXM]
29	iso-tp4	ISO Transport Protocol Class 4	[RFC905,RC77]
30	netblt	Bulk Data Transfer Protocol	[RFC969,DDC1]
31	mfe-nsp	MFE Network Services Protocol	[MFENET,BCH2]
32	merit-inp	MERIT Internodal Protocol	[HWB]
33	dccp	Datagram Congestion Control Protocol	
34	3pc	Third Party Connect Protocol	[SAF3]
35	idpr	Inter-Domain Policy Routing Protocol	[MXS1]
36	xtp	XTP	[GXC]
37	ddp	Datagram Delivery Protocol	[WXC]
38	idpr-cmtp	IDPR Control Message Transport Proto	[MXS1]
39	tp++	TP++ Transport Protocol	[DXF]
40	il	IL Transport Protocol	[Presotto]
41	ipv6	Ipv6	[Deering]
42	sdrp	Source Demand Routing Protocol	[DXE1]
43	ipv6-route	Routing Header for IPv6	[Deering]
44	ipv6-frag	Fragment Header for IPv6	[Deering]
45	idrp	Inter-Domain Routing Protocol	[Sue Hares]
46	rsvp	Reservation Protocol	[Bob Braden]
47	gre	General Routing Encapsulation	[Tony Li]
48	mhrp	Mobile Host Routing Protocol	[David Johnson]
49	bna	BNA	[Gary Salamon]
50	esp	Encap Security Payload	[RFC2406]
51	ah	Authentication Header	[RFC2402]
52	i-nlsp	Integrated Net Layer Security	TUBA [GLENN]
53	swipe	IP with Encryption	[JI6]
54	narp	NBMA Address Resolution Protocol	[RFC1735]
55	mobile	IP Mobility	[Perkins]
56	tlsp	Transport Layer Security Protocol using Kryptonet key management	[Oberg]
57	skip	SKIP	[Markson]
58	ipv6-icmp	ICMP for IPv6	[RFC1883]
59	ipv6-nonxt	No Next Header for IPv6	[RFC1883]
60	ipv6-opts	Destination Options for IPv6	[RFC1883]
61		any host internal protocol	[IANA]
62	cftp	CFTP	[CFTP,HCF2]
63		any local network	[IANA]
64	sat-expak	SATNET and Backroom EXPAK	[SHB]
65	kryptolan	Kryptolan	[PXL1]
66	rvd	MIT Remote Virtual Disk Protocol	[MBG]
67	ippc	Internet Pluribus Packet Core	[SHB]
68		any distributed file system	[IANA]
69	sat-mon	SATNET Monitoring	[SHB]

Decimal	Keyword	Protocol	References
70	visa	VISA Protocol	[GXT1]
71	ipcv	Internet Packet Core Utility	[SHB]
72	cpnx	Computer Protocol Network Executive	[DXM2]
73	cphb	Computer Protocol Heart Beat	[DXM2]
74	wsn	Wang Span Network	[VXD]
75	pvp	Packet Video Protocol	[SC3]
76	br-sat-mon	Backroom SATNET Monitoring	[SHB]
77	sun-nd	SUN ND PROTOCOL-Temporary	[WM3]
78	wb-mon	WIDEBAND Monitoring	[SHB]
79	wb-expak	WIDEBAND EXPAK	[SHB]
80	iso-ip	ISO Internet Protocol	[MTR]
81	vmtp	VMTP	[DRC3]
82	secure-vmtp	SECURE-VMTP	[DRC3]
83	vines	VINES	[BXH]
84	ttp	TTP	[JXS]
85	nsfnet-igp	NSFNET-IGP	[HWB]
86	dgp	Dissimilar Gateway Protocol	[DGP,ML109]
87	tcf	TCF	[GAL5]
88	eigrp	EIGRP	[CISCO,GXS]
89	ospfigp	OSPFIGP	[RFC1583,JTM4]
90	sprite-rpc	Sprite RPC Protocol	[SPRITE,BXW]
91	larp	Locus Address Resolution Protocol	[BXH]
92	mtp	Multicast Transport Protocol	[SXA]
93	ax25	AX.25 Frames	[BK29]
94	ipip	IP-within-IP Encapsulation Protocol	[JI6]
95	micp	Mobile Internetworking Control Pro.	[JI6]
96	scc-sp	Semaphore Communications Sec. Pro.	[HXH]
97	etherip	Ethernet-within-IP Encapsulation	[RFC3378]
98	encap	Encapsulation Header	[RFC1241,RXB3]
99		any private encryption scheme	[IANA]
100	gmtp	GMTP	[RXB5]
101	ifmp	Ipsilon Flow Management Protocol	[Hinden]
102	pnni	PNNI over IP	[Callon]
103	pim	Protocol Independent Multicast	[Farinacci]
104	aris	ARIS	[Feldman]
105	scps	SCPS	[Durst]
106	qnx	QNX	[Hunter]
107	a-n	Active Networks	[Braden]
108	ipcomp	IP Payload Compression Protocol	[RFC2393]
109	snp	Sitara Networks Protocol	[Sridhar]
110	compaq-peer	Compaq Peer Protocol	[Volpe]
111	ipx-in-ip	IPX in IP	[Lee]
112	vrrp	Virtual Router Redundancy Protocol	RFC3768]
113	pgm	PGM Reliable Transport Protocol	[Speakman]
114		any 0-hop protocol	[IANA]
115	l2tp L	ayer Two Tunneling Protocol	[Aboba]
116	ddx D	-II Data Exchange (DDX)	[Worley]
117	iatp I	nteractive Agent Transfer Protocol	[Murphy]
118	stp	Schedule Transfer Protocol	[JMP]
119	srp	SpectraLink Radio Protocol	[Hamilton]

Decimal	Keyword	Protocol	References
120	uti	UTI	[Lothberg]
121	smp	Simple Message Protocol	[Ekblad]
122	sm	SM	[Crowcroft]
123	ptp	Performance Transparency Protocol	[Welzl]
124	isis	over IPv4	[Przygienda]
125	fire		[Partridge]
126	crtp	Combat Radio Transport Protocol	[Sautter]
127	crudp	Combat Radio User Datagram	[Sautter]
128	sscopmce		[Waber]
129	iplt		[Hollbach]
130	sps	Secure Packet Shield	[McIntosh]
131	pipe	Private IP Encapsulation within IP	[Petri]
132	sctp	Stream Control Transmission Protocol	[Stewart]
133	fc	Fibre Channel	[Rajagopal]
134	rsvp-e2e-ignore		[RFC3175]
135	-	Modility Header	[RFC3775]
136	udp-lite		[RFC3828]
137	mpls-in-ip		[RFC4023]
138-252		Unassigned	[IANA]
253		Use for experimentation and testing	[RFC3692]
254		Use for experimentation and testing	[RFC3692]
255		Reserved	[IANA]

Figure A.1: Protocol names known in XORP NAT - long version.

Decimal	Keyword	Protocol name
0	ipv6	IP version 6 (IPv6) in IP
1	icmp	ICMP
2	igmp	Internet Group Management Protocol (IGMP)
4	ipip	IP in IP
6	tcp	TCP
8	egp	External Gateway Protocol (EGP)
17	udp	UDP
46	rsvp	Resource Reservation Protocol (RSVP)
47	gre	Generic routing encapsulation (GR)
50	esp	IPSec Encapsulating Security Payload (ESP)
51	ah	IP Security (IPSec) authentication header (AH)
89	ospf	Open Shortest Path First (OSP)
103	pim	Protocol Independent Multicast (PIM)
112	vrrp	Virtual Router Redundancy Protocol (VRRP)

Figure A.2: Protocol names known in the XORP NAT module - short version.

Sub protocol names and numbers

Sub protocol names and numbers understood by the XORP NAT module is listed in the tables below. Selection of the long or short version is done with the same C++ preprocessor symbol as for the protocol names.

Port Number	Port Name
20	ftp-data
21	ftp
22	ssh
23	telnet
25	smtp
53	domain
65	tacacs-ds
67	bootps
67	dhcp
68	bootpc
69	tftp
79	finger
80	http
88	kerberos-sec
109	pop2
110	pop3
111	sunrpc
113	ident
113	auth
119	nntp
123	ntp
137	netbios-ns
138	netbios-dgm
139	netbios-ssn
143	imap
161	snmp
162	snmptrap
177	xdmcp
179	bgp
389	ldap
434	mobileip-agent
435	mobilip-mn
443	https
444	snpp
464	kpasswd
512	comsat
512	exec
513	login
513	who
514	shell
514	syslog
515	printer
517	talk
518	ntalk
525	timed
543	klogin
544	kshell
639	msdp
993	imaps
1080	socks
1483	afs
1812	radius
1813	radus-acct
2049	nfs
2401	cvspserver

Figure A.3: TCP/UDP port numbers and names known in XORP NAT - long version.

Port Number	Port Name
20	ftp-data
21	ftp
22	ssh
23	telnet
25	smtp
53	domain
67	bootps
67	dhcp
69	tftp
80	http
109	pop2
110	pop3
123	ntp
143	imap
443	https
993	imaps

Figure A.4: TCP/UDP port names and numbers known in XORP NAT - short version.

Glossary

LAN Local Area Network.

NAT Network Address Translation. A method to translate the source and/or destination addresses of IP packet between hosts connected to two networks where the address plan is not coordinated - e.g. a LAN with private IP addresses (as defined in rfc 1597) and the public internet. NAT only translate IP addresses so there must be a one-to-one translation map between hosts on either network. See also NAPT.

NAT administrator The person who is configuring the NAT device.

XORP eXtensible Open Router Platform. A open source project that develops software for an Internet router. The project homepage is: www.xorp.org.

Realm A realm is a separate individual IP address domain, e.g. separate realms are IP address domains, which is not part of the same IP address plan. See also the extended description at the end of this glossary.

NAPT Network Address and Port Translation. NAPT transkates both IP-addresses and TCP or UDP port numbers. NAPT is most often used in dynamic NAT when many IP adresses is translated into one IP address. when there is a conflict e.g. the same source port is used by two inside IP addresses (hosts) the NAPT function translates one of these into a new port number when the IP packet passes the NATP module. Hense the outside server will see two different source ports accessing it from the same IP address.

IPv4 (XORP) : XORP data type that can hold an IP version 4 IP address.

IPv6 (XORP) : XORP data type that can hold an IP version 6 IP address.

IPvX (XORP) : XORP data type that can holdd either an IPv4 or an IPv6 IP address.

MFC: Multicast Forwarding Cache: Another name for an entry in the multicast forwarding engine (typically used on UNIX systems).

MFEA : Multicast Forwarding Engine Abstraction.

MLD/IGMP : Multicast Listener Discovery/Internet Group Management Protocol.

MRIB : Multicast routing Information Base.

PIM-SM : Protocol Independent Multicast–Sparse Mode.

RIB : Routing Information Base.

BGP : Boarder Gateway Protocol.

TCP : Transport Control Protocol.

UDP : Universal Datagram Protocol.

TCP session : TCP session is established by sending a TCP packet with the SYN flag set. packet, and terminated by sending or receiving a TCP packet with the FIN flag set. Packets belonging to the same TCP session is having the same IP source address, source port number, destination IP address and destination port number.

UDP session : UDP does not have a session state, but UDP sessions is often (in NAT terminology) defined by packets having the same IP source address, source port number, destination IP address, and destination port number. As UDP does not have sessions the NAT device often has a timeout for UDP sessions by e.g. not having received packets for the same session in a defined amount of time.

Interface : XORP physical interface of a XORP router. Normally a physical interface is a network interface card (NIC). XORP interfaces is by definition in XORP a name for an interface that the XORP router can identify by that name via e.g. the `ifconfig` UNIX command. In XORP physical interfaces has virtual interfaces as a sub-interface- see "XORP vif".

Vif : (XORP Virtual Interface): XORP distinguishes between interfaces (aka network interface cards) and virtual interfaces which is configurable interfaces for a single e.g. IPv4 or IPv6 network. To give XORP a unified logical appearance of the hardware interfaces is by convention divided up like this. For simpler interfaces (e.g. Ethernet NICs with an IP subnetwork connected directly, the vif will hold the IP definition (IP-address and subnet mask) and the XORP Interface will only be a holder for the vif configuration node. On more advanced Ethernet configurations such as IEEE 802.1Q (Ethernet trunk with a number of Virtual LANs inside), the Interface holds the name of the physical NIC, and a number of vif definitions below the Interface (node) each describe a Virtual LAN interface.

DHCP : Dynamic Host Configuration Protocol. A protocol used to request, allocate and distribute IP configuration data (IP-addresses, subnet mask, default gateway, DNS domain name etc.) to hosts requesting such.

MBGP : Multicast Boarder Gateway Protocol.

OSI : Open Systems Interconnect. ISO network model from the 1980 era.

ISO : International Standards Organization. http://www.iso.org.

NAT terminology definitions

The following 7 quotations is from RFC-2391 [32]. They are the key definitions of the Network Address Translation (NAT) functionality which is the reason to print them here.

TU ports, Server ports, Client ports Quotation from RFC 2391:

> "We will refer TCP/UDP ports associated with an IP address simply as "TU ports".
>
> For most TCP/IP hosts, TU port range 0-1023 is used by servers listening for incoming connections. Clients trying to initiate a connection typically select a TU port in the range of 1024-65535. However, this convention is not universal and not always followed. It is possible for client nodes to initiate connections using a TU port number in the range of 0-1023, and there are applications listening on TU port numbers in the range of 1024-65535.
>
> A complete list of TU port services may be found in Appendix A. The TU ports used by servers to listen for incoming connections are called

> ”Server Ports” and the TU ports used by clients to initiate a connection to server are called ”Client Ports”.”

Session flow vs. Packet flow Quotation from RFC2391:

> “Connection or session flows are different from packet flows. A session flow indicates the direction in which the session was initiated with reference to a network port. Packet flow is the direction in which the packet has traversed with reference to a network port. A session flow is uniquely identified by the direction in which the first packet of that session traversed.
>
> Take for example, a telnet session. The telnet session consists of packet flows in both inbound and outbound directions. Outbound telnet packets carry terminal keystrokes from the client and inbound telnet packets carry screen displays from the telnet server. Performing address translation for a telnet session would involve translation of incoming as well as outgoing packets belonging to that session.
>
> Packets belonging to a TCP/UDP session are uniquely identified by the tuple of (source IP address, source TU port, target IP address, target TU port). ICMP sessions that correlate queries and responses using query id are uniquely identified by the tuple of (source IP address, ICMP Query Identifier, target IP address). For lack of well-known ways to distinguish, all other types of sessions are lumped together and distinguished by the tuple of (source IP address, IP protocol, target IP address).”

Start of session for TCP, UDP and others Quotation from RFC2391:

> “The first packet of every TCP session tries to establish a session and contains connection startup information. The first packet of a TCP session may be recognized by the presence of SYN bit and absence of ACK bit in the TCP flags. All TCP packets, with the exception of the first packet must have the ACK bit set.
>
> The first packet of every session, be it a TCP session, UDP session, ICMP query session or any other session, tries to establish a session. However, there is no deterministic way of recognizing the start of a UDP session or any other non-TCP session.
>
> Start of session is significant with NATs, as a state describing translation parameters for the session is established at the start of session. Packets pertaining to the session cannot undergo translation, unless a state is established by NAT at the start of session.”

End of session for TCP, UDP and others Quotation from RFC2391:

> “The end of a TCP session is detected when FIN is acknowledged by both halves of the session or when either half receives RST bit in TCP flags field. Within a short period (say, a couple of seconds) after one of the session partners sets RST bit, the session can be safely assumed to have been terminated.
>
> For all other types of session, there is no deterministic way of determining the end of session unless you know the application protocol. Many heuristic approaches are used to terminate sessions. You can make the assumption that TCP sessions that have not been used for say, 24 hours, and non-TCP sessions that have not been used for say, 1 minute, are terminated. Often this assumption works, but sometimes it doesn’t.

These idle period session timeouts may vary considerably across the board and may be made user configurable.

Another way to handle session terminations is to timestamp sessions and keep them as long as possible and retire the longest idle session when it becomes necessary."

Basic Network Address Translation (Basic NAT) Quotation from RFC2391:

"Basic NAT is a method by which hosts in a private network domain are allowed access to hosts in the external network transparently. A block of external addresses are set aside for translating addresses of private hosts as the private hosts originate sessions to applications in external domain. Once an external address is bound by the NAT device to a specific private address, that address binding remains in place for all subsequent sessions originating from the same private host. This binding may be terminated when there are no sessions left to use the binding."

Network Address Port Translation (NAPT) Quotation from RFC2391:

"Network Address Port Translation(NAPT) is a method by which hosts in a private network domain are allowed simultaneous access to hosts in the external network transparently using a single registered address. This is made possible by multiplexing transport layer identifiers of private hosts into the transport identifiers of the single assigned external address. For this reason, only the applications based on TCP and UDP protocols are supported by NAPT. ICMP query based applications are also supported as the ICMP header carries a query identifier that is used to corelate responses with requests. Sessions other than TCP, UDP and ICMP query type are simply not permitted from local nodes, serviced by a NAPT router."

Load share Quotation from RFC2391:

"Load sharing for the purpose of this document is defined as the spread of session load amongst a cluster of servers which are functionally similar or the same. In other words, each of the nodes in cluster can support a client session equally well with no discernible difference in functionality. Once a node is assigned to service a session, that session is bound to that node till termination. Sessions are not allowed to swap between nodes in the midst of session.

Load sharing may be applicable for all services, if all hosts in server cluster carry the capability to carry out all services. Alternately, load sharing may be limited to one or more specific services alone and not to others.

Note, the term "Session load" used in the context of load share is different from the term "system load" attributed to hosts by way of CPU, memory and other resource usage on the system."

About the word "Realm"

The extended description of the word "Realm". The description is from the free encyclopedia "Wikipedia".

Realm For other meanings see "Realm (disambiguation)"

A Realm is the dominions of a king (or queen), a kingdom. The Old French reaume (modern French royaume) was the form first adopted in English, and the modern spelling does not appear fixed until the beginning of the 17th century. The word must be referred to a supposed Med. Lat. regalimen, from regalis, of or belonging to a rex, (king).[1]

It is particularly used for those states whose name includes the word Kingdom (for example, the United Kingdom), to avoid clumsy repetition of the word in a sentence. (For example, "The Queen's realm, the United Kingdom...".)

It is frequently used to refer to territories "under" a monarch, yet not a physical part of his or her "kingdom"; for example, the various Commonwealth Realms under the British crown, in Realm of Sweden, or to Holstein that until the Second War on Schleswig was an important part of the Danish king's realm stretching to the border of Hamburg, although not a part of the Danish kingdom. Similarly, the Cook Islands, Niue, and Tokelau are considered parts of the Realm of New Zealand, though they are not part of New Zealand proper. Likewise, the Faroe Islands and Greenland remain parts of the Danish Realm.

Relm (disambiguation) A Realm can mean:

- A Realm, the dominions of a king (or queen), a kingdom.
- The word realm is often used in fantasy books or movies, primarily as a synonym for a world usually other than our own.
- In Java EE realm terms a database containing users, usergroups and their roles (sets of permissions to access server-resources). Optionally a realm manages user-passwords, certificates and authentication logic. Also a realm can refer to a web domain.
- Gaming - in Blizzard Entertainment's newer games (Diablo II. Warcraft III and World of Warcraft), the term realm is used interchangeably with a computer gaming server (some MMORPGs use the term Shard to describe a gaming server).
- Realmz is a fantasy adventure.
- The Realm was the name of a BBS and underground community of computer hackers based in Melbourne, Australia in the late 1980s. Its most notable members were Phoenix, The Force and Electron, who were both arrested by the Australian Federal Police in April 1990 in Australia's first major computer hacker bust.
- Various theories have been supposed which used realms to explain how time moves.[citation needed]

Bibliography

[1] The XORP Project: *XORP Design Overview* XORP technical document. http://www.xorp.org/releases/1.3/docs/design_arch/design_arch.pdf

[2] The XORP Project: *XORP Software Status page* XORP website page. http://www.xorp.org/status.html

[3] The XORP Project: *XORP BGP Routing Daemon.* XORP technical document. http://www.xorp.org

[4] The XORP Project: *XORP Forwarding Engine Abstraction.* XORP technical document. http://www.xorp.org

[5] The XORP Project: *XORP Inter-Process Communication Library overview.* XORP technical document. http://www.xorp.org

[6] The XORP Project: *XORP MLD/IGMP Daemon.* XORP technical document. http://www.xorp.org

[7] The XORP Project: *XORP Multicast Forwarding Engine Abstraction.* XORP technical document. http://www.xorp.org

[8] The XORP Project: *XORP Multicast Routing Design Architecture.* XORP technical document. http://www.xorp.org

[9] The XORP Project: *XORP PIM-SM Routing Daemon.* XORP technical document. http://www.xorp.org

[10] The XORP Project: *XORP Router Manager Process (rtrmgr).* XORP technical document. http://www.xorp.org

[11] The XORP Project: *XORP Route Information Base (rib).* XORP technical document. http://www.xorp.org

[12] The XORP Project: *XORP XRL Interfaces: Specifications and Tools.* XORP technical document. http://www.xorp.org

[13] The XORP Project: *XORP User Manual Version 1.3* XORP User Manual. http://www.xorp.org

[14] The XORP Project: *XORP SNMP Agent* XORP User Manual. http://www.xorp.org

[15] The XORP Project: *An Introduction to Writing a XORP Process* XORP project documentation. http://www.xorp.org/releases/1.3/docs/xorpdev_101/xorpdev_101.pdf

[16] The XORP Project: *Decoupling Policy from Protocols: Implementation Issues in Extensible IP Router Software* Andrea Bittau, Mark Handley University College London XORP papers. http://www.xorp.org/papers/policy.pdf

[17] Mark Handley, Orion Hodson, and Eddie Kohler. XORP: An Open Platform for Network Research. In *Proceedings of HotNets-I Workshop*, Princeton, New Jersey, USA, October 2002. (This document is not referenced in the report)

[18] The Click Modular Router Project. *The Click Modular Router Project homepage* http://www.read.cs.ucla.edu/click/.

[19] Eddie Kohler, *"The Click Modular Router" Ph.D thesis* M.I.T, february 2001, http://pdos.csail.mit.edu/papers/click:kohler-phd/thesis.pdf

[20] Eddie Kohler, Robert Morris, Massimiliano Poletto, *"Modular Components for Network Address Translation", Proceedings of OPENARCH '02, New York, June 2002, pages 39-50.* ICSI Center for Internet Research, MIT Lab for Computer Science, Mazy Networks, http://pdos.csail.mit.edu/papers/rewriter-openarch02.pdf

[21] J. Postel, *"User Datagram Protocol" (UDP), RFC 768,* ISI, August 1980.

[22] DARPA Internet program - protocol specificaqtion *"TRANSMISSION CONTROL PROTOCOL" (TCP), RFC 793,* ISI, September 1981.

[23] Linda J. Seamonson, Eric C. Rosen, *""STUB" Exterior Gateway Protocol" (EGP), RFC 888,* BBN Communications, January 1984.

[24] Fuller, V., Li, T., and J. Yu, *"Classless Inter-Domain Routing (CIDR) an Address Assignment and Aggregation Strategy", RFC 1519,* BARRNet, cisco, Merit, OARnet, September 1993. (This document is not referenced in the report)

[25] Rekhter, Y., Moskowitz, B., Karrenberg, D., and G. de Groot, *"Address Allocation for Private Internets", RFC 1597,* T.J. Watson Research Center, IBM Corp., Chrysler Corp., RIPE NCC, March 1994.

[26] K. Egevang, P. Francis, *"The IP Network Address Translator (NAT)", RFC 1631,* Cray Communications, NTT, May 1994. (This document is not referenced in the report but is a very early NAT rfc document)

[27] J. Reynolds, J. Postel, *"Assigned Numbers", RFC 1700,* (Now Obsoleted by RFC 3232) ISI, October 1994. (This document is not referenced in the report)

[28] J. Perkins, *"IP Encapsulation within IP", RFC 2003,* IBM, April 1998. http://www.rfc-editor.org/rfc/rfc2003.txt

[29] Rekhter, Y., Moskowitz, B., Karrenberg, D., and G. de Groot, *"OSPF Version 2", RFC 2328,* J. Moy Ascend Communications, Inc. April 1998. ftp://ftp.rfc-editor.org/in-notes/rfc2328.txt

[30] G. Malkin Bay Networks *"RIP Version 2", RFC 2453,* November 1998. ftp://ftp.rfc-editor.org/in-notes/rfc2453.txt

[31] P. Srisuresh, M. Holdrege, *"IP Network Address Translator (NAT) Terminology and Considerations" RFC 2663,* Lucent Technologies, August 1999 http://www.rfc-editor.org/rfc/rfc2663.txt

[32] P. Srisuresh, D. Gan *"Load Sharing using IP Network Address Translation (LS-NAT), RFC 2391,"* Lucent Technologies, Juniper Networks, Inc., August 1998

[33] T. Bates, Y. Rekhter Cisco Systems, R. Chandra Redback Networks Inc D. Katz Juniper Networks *"Multiprotocol Extensions for BGP-4" RFC 2391,* June 2000. http://www.rfc-editor.org/rfc/rfc2858.txt

[34] J. Reynolds, *"Assigned Numbers: RFC 1700 is Replaced by an On-line database", RFC 3232,* RFC Editor, January 2002.

[35] Y. Rekhter, T. Li, S. Hares. *"Multiprotocol Extensions for BGP-4" RFC 4271,* January 2006. ftp://ftp.rfc-editor.org/in-notes/rfc4271.txt

[36] *"Protocol Numbers" Iana database,* October 2006. http://www.iana.org/assignments/protocol-numbers

[37] *"Port Numbers" Iana database,* October 2006. http://www.iana.org/assignments/port-numbers

[38] *"ICMP TYPE NUMBERS" Iana database,* July 2006. http://www.iana.org/assignments/icmp-parameters

[39] Cisco technical documentation *"Configuring Network Address Translation: Getting started", Document ID: 13772a* http://www.cisco.com/en/US/tech/tk648/tk361/technologies_tech_note09186a0080094e77.shtml

[40] Juniper JUNOS Technical Documentation *"STM Internet Software for J-seriesTM, M-seriesTM, and T-seriesTM Routing Platforms Services Interfaces Configuration Guide. Release 7.5"*, http://www.juniper.net/techpubs/software/junos/junos75/swconfig75-services/download/nat-config.pdf