VeriNeC Firewall

MSc Thesis: A Firewall Implementation for the VeriNeC Simulator

Department of Informatics
University of Fribourg, Switzerland

Author:
Jason Hug
Rue de Lausanne 51
1700 Fribourg
jason.hug@unifr.ch

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

20th October 2006

Abstract

In this report the implementation of a firewall for the Verified Network Configuration (VeriNeC) *Simulator* is discussed. The firewall simulates real life counterparts with a *Packet-Filter* and *Stateful Inspection*. Both components, which are based upon the *IPTables* implementation, were successfully integrated within the VeriNeC *Simulator*. How this was accomplished is described in this master thesis.

Keywords: VeriNeC, Packet Filter, IPTables, Stateful Inspection, Network Simulation

Contents

Contents **2**

1 Introduction **4**
1.1 Objective and Overview . . . 4

2 The VeriNeC Project **6**
2.1 Introduction . . . 6
2.2 Architecture . . . 6
2.2.1 Network Definition . . . 6
2.2.2 Verification . . . 8
2.2.3 Distribution . . . 11

3 Firewall **12**
3.1 Introduction . . . 12
3.2 IPTables . . . 13
3.3 Packet Filter . . . 14
3.3.1 Rules . . . 15
3.3.2 Actions . . . 16
3.3.3 Policies . . . 16
3.3.4 Example . . . 17
3.4 Stateful Inspection . . . 18
3.4.1 Conntrack . . . 18
3.4.2 Match-State . . . 23
3.5 VeriNeC's Packet Filter Schema . . . 24

4 Implementation **26**
4.1 Introduction . . . 26
4.2 Packet-Filter . . . 26
4.2.1 Rules . . . 29
4.2.2 Actions . . . 32
4.2.3 Policies . . . 33
4.3 Stateful Inspection . . . 35
4.3.1 The State Table . . . 36
4.3.2 Workflow . . . 37
4.3.3 Timeout . . . 39
4.3.4 Extending . . . 40
4.3.5 Shortcomings . . . 42

4.4 Logging . . . 42

5 Conclusion 44

Bibliography 46

A Acronyms 48

B Verinec Schemas 50
B.1 Network Topology Schema from network.xsd . . . 50
B.2 Node Schema from node.xsd . . . 51
B.3 Packet-Filter Schema from node.xsd . . . 52
B.3.1 packet-filters . . . 52
B.3.2 default-policy . . . 53
B.3.3 packet-filter-rule . . . 54
B.3.4 packet-match-list . . . 55
B.3.5 packet-action-list . . . 57
B.4 Extendended Events Schema from events.xsd . . . 57

C Examples 61
C.1 Properties file for Stateful Inspection . . . 61
C.2 Network Definition . . . 61
C.2.1 Complete Network Definition . . . 61

Chapter 1

Introduction

Network administration can be a tedious job in a heterogeneous network-environment. Each computer, router, firewall and service needs to be configured in a safe and correct manner. Not only does the first configuration have to be manually deployed on each network component but also altering the settings on one component may imply the alteration of all other components in the network. As one can see, managing such a given system can be complicated and very time consuming, especially when one considers how networks grow larger and larger each day and each network component needs to be configured with its own administration tool.

The project VeriNeC [1] aims to simplify network configuration and administration. The project is funded by the Swiss National Science Foundation, and has been in development since September 2004. VeriNeC achieves its task with three core modules. The first module helps the user to either define a new network or gather information about an already existing network (*Network Definition*). The second module validates the correctness of a given network configuration using a *Simulator* (*Verification*). And the third module writes the configuration to each component on the network (*Distribution*). The three modules use one unified description of the network topology and component configuration with the help of eXtensible Markup Language (XML).

1.1 Objective and Overview

In this master thesis the main goal was to introduce a packet-filtering firewall into the VeriNeC *Simulator*. VeriNeC contains a module which simulates the behavior of a network configuration. The simulation result is then stored in a well defined XML document. Different services and protocols have been implemented into the *Simulator*, but a packet-filtering firewall was missing. The task of this thesis was to implement a packet-filtering firewall, configurable through VeriNeC, and to extend the output log file in such a way that the firewall events are logged. The firewall would have to be implemented in such a way that it depicts real life counterparts that are used in todays computer networks.
The motivation to include a firewall component into the *Simulator* includes the fact that most networks (private or corporate) these days have some sort of packet-filtering firewall present. This may be a dedicated hardware firewall component or one which is installed as a software product on a local machine. When validating the abstract definition of a network, one needs to consider the fact of such a configured component which might

hinder network communication (or let unwanted communication pass through) before distributing the configuration onto the real network. Without these tests the network may then not work as expected or even worse open security holes for unwanted guests.

The task at hand was completed over a time frame of eight months. This report, which is a part of the master thesis, describes the process of achieving the integration of a packet-filtering firewall into the existing VeriNeC *Simulator*.
This report can be divided into four parts. The first part gives a short overview of VeriNeC and its components. The second part describes the fundamentals of a packet-filtering firewall. The third part describes how the firewall was implemented and lists all of it's features. And the last part concludes this report.

Chapter 2

The VeriNeC Project

2.1 Introduction

As networks grow larger and larger the job of administrating them gets harder and harder. Not only does each network component come with its own administration tool but also configuring all components in such a way that smooth network operation is guaranteed can be a difficult task. The VeriNeC project's goal is to introduce a tool which simplifies network administration.
In this chapter a rough overview of VeriNeC is given. We will look at the project's architecture and its features will be discussed briefly.

2.2 Architecture

The core architecture of VeriNeC is composed of three modules. Each module accesses the setup of a given network through an abstract *Network Definition Document* which is stored as an XML document. The idea behind the abstraction is to introduce a common language which expresses the network topology and its components configurations. Configuring services is difficult to accomplish due to the heterogeneous attribute of networks. Each component of the network may be configured in a different fashion depending on operating system or manufacturer. Their functionality however stays the same. Hence the idea of introducing this abstraction.

The modules each have a specific task based around the *Network Definition Document* which one can see in Figure 2.1. The first module which consists of the *Editor* and the *Importer* is responsible for creating a valid *Network Definition Document* (see Section 2.2.1). The *Verification* module (see Section 2.2.2) tests whether the given *Network Definition Document* satisfies specific requirements. And the *Distribution* module (see Section 2.2.3) is responsible for translating the abstract service configuration into real life configuration syntax which is then deployed onto the specified system [6].

2.2.1 Network Definition

The *Network Definition Document* describes the network topology and the configuration of its components (nodes) in an abstract manner. VeriNeC provides tools which help create such a definition. The *Editor* can create new *Network Definition Documents* of a

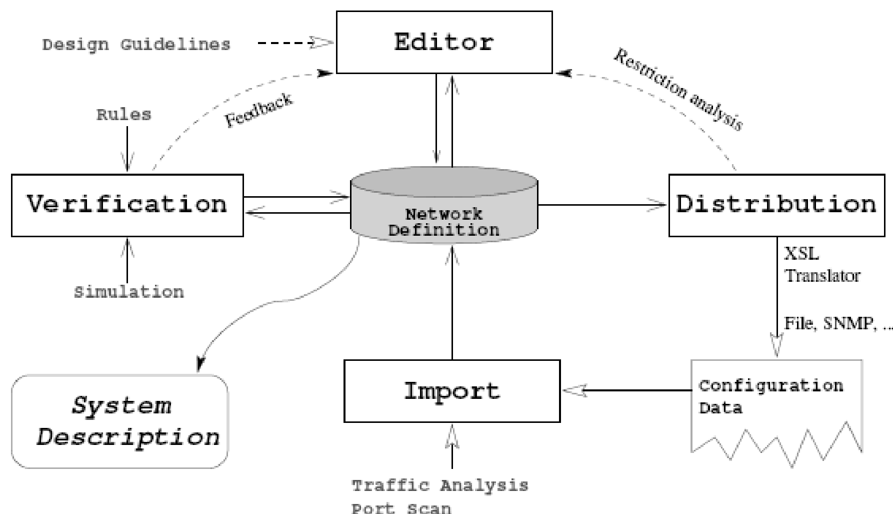


Figure 2.1: Architecture of VeriNeC [6]

network or may edit already created ones. The *Import* module creates the *Network Definition Document* based on locally stored configuration files [9] of network components and makes use of a *Network Scanner* [8] to determine the topology of the given network. The topology and the configuration of all nodes are defined in two separate XML documents. The network topology document is used for documentation and for simulation. It describes how a Network Interface Card (NIC) is interconnected within the given system. Listing 2.1 gives an example of two network topologies. The first one named 'intranet' consists of three nodes which are connected with each other. The second network named 'extranet' consits of two interconnected nodes. The corresponding XML Schema can be found in Appendix B.1. Furthermore, a complete *Network Definition* example is provided in Appendix C.2.1.

```
1  <networks>
2    <network name="intranet">
3      <connected binding="1.1.335"/>
4      <connected binding="1.1.184"/>
5      <connected binding="1.1.193"/>
6    </network>
7    <network name="extranet">
8      <connected binding="1.1.237"/>
9      <connected binding="1.1.034"/>
10   </network>
11 </networks>
```

Listing 2.1: Network Topology Example [2]

The configuration of all nodes is also described in an XML document. A node may represent a workstation, server, switch, router or hardware firewall. As well as describing the physical configuration of the node (e.g. its NIC), the relation to each connected network is also specified. Further, services running on the node are also described here. At the time of writing, VeriNeC has implemented the following services, which the *Simulator*

does not yet consider:

- Routing
- Domain Name System (DNS)
- Dynamic Host Configuration Protocol (DHCP)
- Packet-Filters (Service which has been implemented in the MSc project described in this thesis.)

Listing 2.2 shows an example of a configuration of one node. The node is configured with one Ethernet NIC which owns two Internet Protocol (IP) addresses with their corresponding DNS servers and gateways. Further in this example no service has been configured. The corresponding XML Schema can be reviewed in Appendix B.2

```
1  <vn:node hostname="diufpc55">
2    <vn:hardware>
3      <vn:ethernet name="GBit Connection" hwaddress="00:10:ab:12:ff:23"
          >
4        <vn:ethernet-binding name="eth0" id="1.1.355">
5          <vn:nw id="i1" address="192.168.0.1" subnet="255.255.0.0"
6           gateway="192.168.0.24" type="ip">
7            <vn:nw-dnsserver ip="192.168.0.254" />
8            <vn:nw-dnsserver ip="192.168.0.253" />
9          </vn:nw>
10         <vn:nw id="i2" address="134.21.9.48" type="ip" />
11       </vn:ethernet-binding>
12     </vn:ethernet>
13   </vn:hardware>
14
15   <vn:services>
16         <vn:routing />
17         <vn:dns />
18         <vn:dhcp />
19         <vn:packet-filters />
20   </vn:services>
21 </vn:node>
```

Listing 2.2: Node Configuration Example [2]

2.2.2 Verification

In this section we will primarily look at the *Simulator* which is part of the *Verification* step. The *Verification* technique will not be discussed here but can be read about in [3]. The role of the *Simulator* is to analyze network behavior and increase one's confidence in the correctness of the configured nodes before distribution. To accomplish this, a framework based on Discrete-Event Modeling and Simulation in Java (DESMO-J) [20] was used to test the network configuration. The *Simulator* builds a virtual network with virtual nodes from the abstract *Network Definition Document*. Each node is connected

over one or several NICs to one or several virtual networks. Each node contains layers [3] which offer network services. The layer architecture is related to the one described in [11]. The basic idea behind this approach is that each layer offers a network service to the above layer. Network protocols (i.e Transport Control Protocol (TCP) or IP) then implement the services. Figure 2.2 shows an example node in the *Simulator* with its corresponding services.

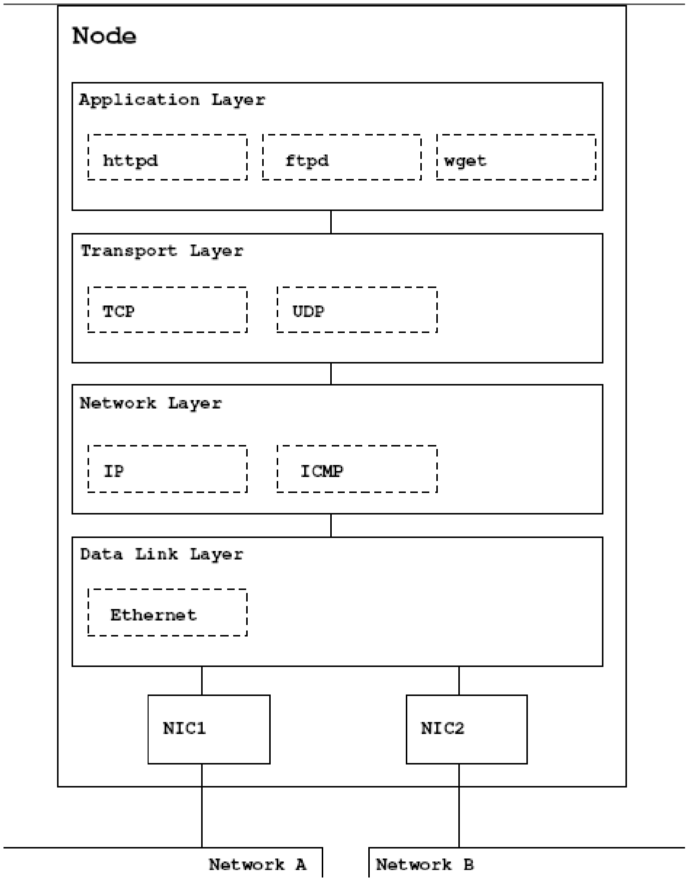


Figure 2.2: Node in the VeriNeC Simulator [2]

The network is configured through the *Network Definition Document*. When instantiating the *Simulator*, the framework instantiates the network with its nodes and connects each node to the network. Depending on how the nodes were configured in the *Network Definition Document* the corresponding layers and services are also instantiated. Further, the *Simulator* needs some sort of input to start the simulation. Each event may send, receive drop or route a packet, launch or finish an application [3]. An event may also trigger another event.
The simulator's framework schedules all *Input Events* and triggers them at the specified time. During the simulation all events are logged into a log file which is also XML based. Listing 2.4 shows an example output of a finished simulation run triggered from the *Input Events* from Listing 2.3. As stated above, scheduled events may trigger other events, therefore root elements are initial input events whereas child elements are triggered events caused by a parent event [3]. In this example a client would like to access the web page *test.html*. This event triggers another event which does a DNS lookup which associates the domain name *diuflx02* to its IP address. This process triggers another event which

creates a User Datagram Protocol (UDP) packet which will be sent over the virtual network. Of course this triggers another event since all the layers of the node have to be traversed until a data frame actually is sent to the target node.

```
1 <events>
2  <event time="1" node="diuflx01" layer="5" service="application" id="
      unique1">
3   <application program="wget" parameters="http://diuflx02/test.html"
       type="launch"/>
4  </event>
5  <event time="2" node="diufpc55" layer="5" service="application" id="
      unique2">
6   <application program="wget" parameters="http://diuflx02/test.html"
       type="launch"/>
7  </event>
8 </events>
```

Listing 2.3: Example of an Input Events File [3]

```
1  <events>
2   <event time="0" node="diufpc55" layer="5" service="application" src=
       "diufpc55" dst="diuflx01" id="unique1">
3   <application program="wget" parameters="http://diuflx02/test.html"
       type="launch"/>
4    <event time="1" node="diufpc55" layer="5" service="application" id=
        "unique1">
5     <application program="dns" type="lookup" parameters="diuflx02"/>
6      <event time="2" node="diufpc55" layer="4" service="udp" id="
          unique1" packetid="dns1" src="134.21.3.8" dst="134.21.6.8">
7      <udp type="packetsend" srcport="45401" dstport="53"/>
8       <event time="2" node="diufpc55" layer="3" service="ip" id="
           unique1" packetid="dns2" src="134.21.3.8" dst="134.21.6.8"
           interface="slkjadsmf">
9       <ip type="send" ttl="255" protocol="17"/>
10       ...
11 </events>
```

Listing 2.4: A logfile of a HTTP client causing a DNS lookup [2]

Figure 2.3 recapitulates of what we have seen from the *Simulator* up to this point. The *Network Definition Document* and some *Input Events* are needed so that network behavior can be simulated and logged to the *Log File*.

The later *Verification* step involves the evaluation of the *Simulator's Log File* to find unwanted network behavior. Unwanted network behavior could, for example, arise through misconfiguration of the firewall, which leads to unwanted traffic being possible on the tested network. To achieve this, semantic tests on the content of the *Log File* [5] are performed. These tests link certain unwanted network behaviors to configurations within the *Network Definition Document* [4], so that the user may remedy the malformed configurations.

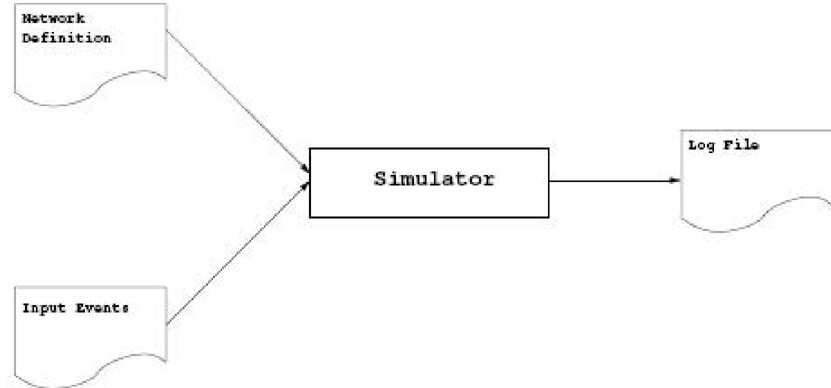


Figure 2.3: Architecture of the VeriNeC Simulator [2]

2.2.3 Distribution

After creating a network setup in abstract form and thoroughly testing it, it is time to distribute the configuration to all physical network components that are part of the defined network. As already stated in the introduction to this report, doing this by hand would not only be tedious but also error prone. It is VeriNeC's goal to provide a tool which distributes the configuration of each node defined in the *Network Definition Document* automatically. This means, there is a process which has to translate the abstract definition of a node's configuration to the specific syntax of the corresponding component. This is where some problems may arise between specific vendor syntax of component configurations and those specified in the abstract language defined in the *Network Definition Document*. Some components may not support everything which can be described by the abstract XML language. Therefore the distribution of each component´s configuration is done in three steps [7]:

1. Restriction: Identifies features that are not supported by a certain implementation.
2. Translation: Generate configuration syntax appropriate from the *Network Definition Document* for the chosen component.
3. Distribute: Actually configures the component.

This report will not go into further details of how these steps have been implemented. The reader may consult [6] for further reading. The reader may like to consult [10] which describes a possible method of remotely configuring a computer[1] using the Java Application Programming Interface (API) for VeriNeC.

[1] Running under the Windows operating system

Chapter 3

Firewall

3.1 Introduction

The term firewall depicts a wall that is supposed to protect an entity from a fire. To bring this depiction into network terminology, a firewall's function is to protect certain assets that are found on point A from point B. In most cases the 'to be protected' asset is found on a Local Area Network (LAN) or on a computer itself. In a more general term a firewall should protect a private network and its assets from outside intrusion which could harm or modify the asset [14]. Figure 3.1 shows how the assets of the *Trusted Network* which includes *Machine A* are protected through a firewall system from *Machine B* or generally speaking from an *Untrusted Network* like the Internet. This security is warranted by controlling the network traffic between the two network parties.

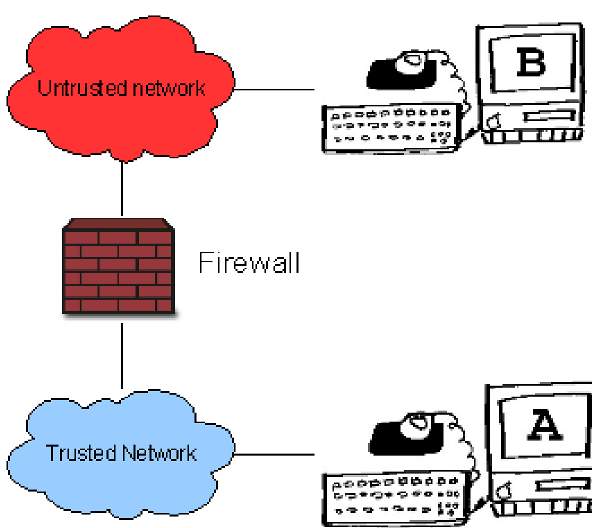


Figure 3.1: Concept of a Network Firewall

Without a firewall, assets found on a *Trusted Network* would be exposed to network attacks. Network attacks may manifest themselves in different ways. In general they try to breach one of the following security concepts [12]:

- Confidentiality: Ensures that information can only be accessed by the party that is

allowed to access it.

- Authentication: Ensures the identity of a party.
- Integrity: Ensures that data is valid and complete.
- Availability: Determines how well a service, running on a network, is accessible.

A firewall is a component which may protect a network from some of these attacks, namely ensuring availability and confidentiality. Firewalls, however, can not guarantee all network security aspects [15], hence the need for other products (Antivirus programms, Intrusion-Detection System (IDS), etc.) which ensure other security aspects.

Firewalls are not standardized. A firewall system summarizes different kinds of technologies, implementations and areas of usage. All kinds of literature ([11, 13, 14, 15]) of firewall concepts make reference to these used technologies but each vendor has its own specification of implementing these concepts ([14], p.17). Therefore it was chosen to study the behavior of one firewall implementation to then later lean on when implementing a firewall for the VeriNeC *Simulator*. This firewall implementation is called *IPTables*[1]. Even though *IPTables* is only the user space tool to administrate packet-filtering rules or Network Address Translation (NAT) rules in a Linux distribution, the name is often used to reference the whole infrastructure[2], which consists of a *Packet Filter* module, connection tracking module and a NAT module[3].

As we have seen, a firewall is a product which may incorporate many different technologies and setups to incorporate network security. In this chapter we will have a look at two concepts of firewall strategies which were implemented for the VeriNeC *Simulator* in the MSc project described here. These are *Packet Filter* and *Stateful Inspection*. But before we delve into these two subjects, we will have a look at *IPTables'* architecture and how it handles network traffic. The last section will explain further features of VeriNeC's firewall by having a look at its *Network Definition* schema.

3.2 IPTables

This section will discuss how packets traverse the firewall architecture *IPTables*. As of Linux Kernel 2.4.0 packet filtering and NAT were pooled to one concept, which can be configured with the `iptables` command. With the help of this tool, rules can be formulated, with which decisions are made based upon the header-information of each network packet. These rules are bundled in so called *chains*. In *IPTables* some of these chains are predefined (see Table 3.1). Other chains can be constructed by the user. These chains are then grouped in *tables*. The table 'nat', for example, contains chains for NAT. While the table 'filter' holds chains which are used for the *Packet Filter*.

Figure 3.2 shows how a packet traverses the *IPTables* infrastructure. When a packet arrives at the network-interface, the rules in the PREROUTING chain of the 'nat' table are consulted. These rules may change destination address and port of an incoming

[1]IPTables, `http://www.netfilter.org/projects/iptables/index.html`
[2]Wikipedia, IPTables, `http://en.wikipedia.org/wiki/Network_address_translation`
[3]There are more modules which can be dynamically loaded into *IPTables* if needed [16]

Table	Default Chains
nat	PREROUTING, OUTPUT, POSTROUTING
filter	INPUT, OUTPUT, FORWARD

Table 3.1: Default Chains for *IPTables* [16]

packet. Next a routing decision is made. If the packet is destined for the machine itself the rules found in the INPUT chain of the 'filter' table are consulted. These rules decide if the packet is accepted or if it should get dropped. If the packet is accepted it is forwarded to the destination process running on the machine. Packets that are not destined for the local machine traverse the FORWARD chain of the 'filter' table.

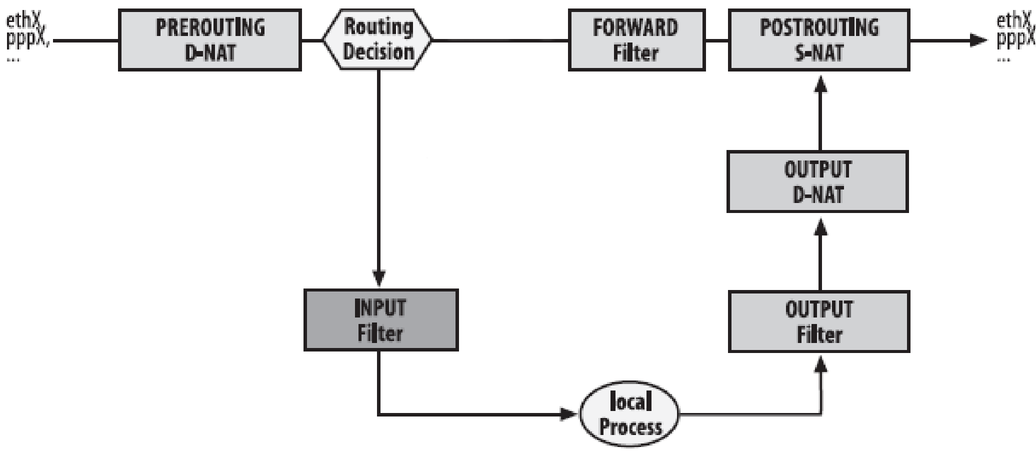


Figure 3.2: Packets traversing *IPTables* [13]

Packets that are being sent from a local process are checked in the OUTPUT chain of the 'filter' table. If the packet is accepted, the OUTPUT chain of the 'nat' table may alter the destination address and port of the packet. In the end, all outgoing packets traverse the POSTROUTING chain of the 'nat' table where source address and port may be altered. With this method one can achieve masquerading [13]. All outgoing packets are manipulated in such a way that the packets seem to come from the firewall. This method is mostly used to hide the internal network from the Internet, since public IPv4 ranges are becoming more and more scarce as the Internet grows.

3.3 Packet Filter

All firewalls that are used on todays software or hardware driven firewall solutions have some sort of a packet-filtering module present. Packet filtering is a method to filter out packets based upon their protocol's header information. These headers may contain the following information which firewall rules are based upon:

- Protocol Type (Internet Control Message Protocol (ICMP), UDP, TCP, etc.)
- Source Address (IP)

- Destination Address (IP)
- Source Port (TCP, UDP)
- Destination Port (TCP, UDP)
- Flags (particularly the ones found in TCP)

As already seen in Section 3.2, rules are grouped in chains. For the packet-filtering module in *IPTables* the default chains INPUT, OUTPUT and FORWARD are used. In this section we will have a look at some basic packet-filtering rules and policies which are specified by *IPTables* [16]. Please note, however, that this section does not completely describe *IPTables'* funcionality. The most important features, which were later implemented into the *Simulator*, are shown here. For a complete manual on *IPTables*, please consult its manpage [17].

3.3.1 Rules

A rule consists of patterns and an action. Patterns define settings a packet must contain so that the action is triggered. In this section we will have a look at some basic patterns one may define in *IPTables*:

- Protocol (`-p [!] protocol`[4]): This pattern defines the type of `protocol` (TCP, UDP, ICMP, etc.) a packet must belong to so that the rule applies.
- Address: This pattern defines the source (`-s [!] address[/mask]`[4]) or destination (`-d [!] address[/mask]`[4]) address and mask. Therefore the address range is given in Classless Inter-Domain Routing (CIDR)-notation[5].
- Interface: With this pattern one can define from which interface a packet originates and in which direction it is heading (outgoing: `-o [!] interface`[5], incoming: `-i [!] interface`[5]) for that the rule applies.

Note, that the above and the following patterns can be negated with a `!`, which would inverse the pattern-match (i.e. `-p ! TCP` would apply for all non-TCP packets).

Protocol-Specific Rules

To enhance the depth of the rules, one may define protocol specific patterns which are used for packets of that protocol. For example: If TCP pattern was selected the rule can be extended by specifying which source port (e.g. `-p TCP --sport 8080`) the packet contains. Some further rules:

- TCP or UDP packets which match a certain source (`--sport [!] port[:port]`[5]) or destination (`--dport [!] port[:port]`[5]) port or port-range.

[4]Corresponding IPTables commands [17]
[5]Wikipedia on CIDR, `http://en.wikipedia.org/wiki/Classless_Inter-Domain_Routing`

- TCP-header flags that are set or not (`[!] --tcp-flags mask comp`[5]). The first argument `mask` defines which flags should be examined. The second argument `comp` defines which flags have to be set. For example one may define that a TCP-packet, where the SYN flag was set and the RST unset (`--tcp-flags SYN, RST SYN`), would apply to a rule. The following flags can be examined: SYN, ACK, FIN, RST, URG, PSH, ALL (all flags) or NONE (no flags).

- Specific ICMP-types (`--icmp-type [!] typename`[5]). The `typename` can be either specified by its corresponding numeric value or name. Available ICMP types can be found by typing `iptables -p icmp -h` in the console of any Unix or Linux distribution that has *IPTables* installed.

3.3.2 Actions

If a rule applies to a packet passing through the firewall an action is triggered. These actions are defined when specifying the rule. The following actions are available in *IPTables*:

- ACCEPT: The packet is allowed to pass.

- DROP: The packet is blocked.

- RETURN: The current chain seizes to check the packet and returns it to the calling chain, for further consultation.

- LOG: The packet is logged in the system protocol. One can configure which log information should be stored (TCP and/or IP header information) and with which priority (Level). If a log action was defined the chain continues to check the other rules for packet analysis.

- REJECT: The packet is blocked and an ICMP error message is sent back to sender. One may specify what kind of message should be sent back to the sender (available messages: `icmp-net-unreachable`, `icmp-host-unreachable`, `icmp-port-unreachable` (default), `icmp-proto-unreachable`, `icmp-net-prohibited` or `icmp-host-prohibited`).

If none of the rules applied within a chain it simply returns to the caller, or for default-chains the policy is executed.

3.3.3 Policies

In *IPTables* each default chain (INPUT, OUTPUT and FORWARD) has a default action. These default actions are called policies and apply when no rule in the chain was satisfied. The following policies apply for the default chains:

- ACCEPT: The packet is allowed to pass.

- DROP: The packet is blocked.

In *IPTables* self-defined chains do not contain a default policy ([13], p. 289). If no rule applied in the chain the calling rule is consulted to decide what there is to do with the packet.
When configuring a firewall one usually sets the default chains' policies to DROP.

3.3.4 Example

In this Section we will look at an example to see the functionality of a basic *Packet Filter*.

```
1 > iptables -P INPUT DROP
2 > iptables -A INPUT -p TCP --dport 80:100 -j ACCEPT
3 > iptables -A INPUT -s 192.168.0.0/16 -j REJECT
```

Listing 3.1: Packet Filter Example

Listing 3.1 shows three commands which configures *IPTables* to do the following:

- The first line tells the *Packet Filter´s* INPUT chain to use the DROP policy. This action is executed if none of the rules applied.

- The second line creates the first rule which lets all TCP packets through the firewall, that have the destination port-range 80-100.

- The third line creates the second rule which drops packets that contain a source IP address between 192.168.0.1 and 192.168.255.255. It also sends back and ICMP-message to the sender of the packet stating that the port is unreachable.

When a packet arrives at the firewall, depending on the routing decision and in which direction the packet is heading, the corresponding chain is selected to check the rules. Let us assume for this example, that the packet is incoming. This would imply, that the packet would traverse our defined INPUT chain as shown in Figure 3.3.

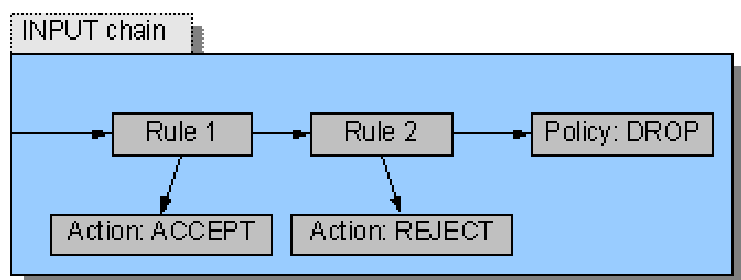


Figure 3.3: Packets traversing the *INPUT chain*

Each rule is checked against the header information of the packet. If for example the first rule would apply, then its corresponding action (in this case ACCEPT) would be executed and the rule checking would cease. If the first rule would not apply, then the second rule is consulted. If the packet applies to the second rule, the packet would be dropped and an ICMP-message would be sent back to the sender (REJECT action). If none of the header-information matches against the defined rules, the default policy is executed and the packet would be dropped. In short: Only TCP-packets that have a destination port between 80-100 may pass the firewall.

3.4 Stateful Inspection

A common problem with static *Packet Filters*, is the fact that they do not know in what state a connection is in. This may cause problems for applications, where it is not known, beforehand, which TCP port the application will use. The File Transfer Protocol (FTP) in active mode[6] is a good example which shows this problem, since it not only uses one active TCP connection but two. The first connection is needed to exchange FTP related commands from client to server, while the second connection is needed for the actual data-transfer. The second connection is established from the server to the client and always uses a different TCP port. The client, therefore, would need to configure the *Packet Filter* with a whole range of incoming open ports, so that the FTP service could take place. Hence not only FTP could use these ports but also other unwelcome applications [15].

To close this gap most firewalls these days have some sort of module present which makes firewall-decisions based upon the state of a network connection. This is called *Stateful Inspection*. The firewall saves all IP based connections in a table and refers to the saved information when deciding if a packet is allowed to pass or not. One of the greater advantages of this method, is the fact that one does not need to care about opening certain ports for response packets, since *Stateful Inspection* can relate these packets to an already active connection. This is especially handy for TCP and UDP[7] protocols but may also be used for other low-level protocols based upon their IP-header[7] [15].
The table entries of existing connections usually contain information about the source and destination addresses plus ports, if present. The states can also be determined by certain header-settings of some protocols. For example, with TCP, certain flags that were set determine in which state a TCP connection is in.

This Section will have a look at *IPTables'* implementation of *Stateful Inspection*. The first part will talk about how network connections are tracked within the Linux Kernel, while the second part will have a look at how this connection tracking can be used within *IPTables*.

3.4.1 Conntrack

In Unix based operating systems, connection tracking is done by a special module within the Kernel. This module is called *Conntrack*. *Conntrack* stores all active network IP-connections' attributes in a table and provides the gathered information to *IPTables*, which can use it to decide whether packets are allowed to pass the firewall or not. The *Conntrack* entries are recalculated each time a packet passes the PREROUTING or the OUTPUT chain of *IPTables* [16].

Conntrack can also be extended with modules for specific network protocols (TCP, UDP and ICMP among others). These modules grab specific information out of the header of a certain protocol, which is then provided to *Conntrack*. Before we describe how the modules operate and which information is collected, let us have a look at a typical *Conntrack* entry. When *Stateful Inspection* has been activated within *IPTables*, the active IP-connections are stored under `/proc/net/ip_conntrack`. Listing 3.2 shows a possible

[6]Wikipedia on FTP, `http://en.wikipedia.org/wiki/Ftp`

[7]Even though UDP and IP are connectionless protocols, *Conntrack* can assign a state to their connections based upon which packets have been sent from sender to receiver. See page 19 for more information.

entry that could be found in this file. With this particular entry, we are dealing with a TCP packet which was sent out with the SYN flag set. This is the initial packet for the three-way handshake [11], which is used to establish a TCP connection between two network parties.

```
1 tcp 6 111 SYN_SENT src=192.168.217.129 dst=192.168.217.130 sport
    =32774 dport=22 [UNREPLIED] src=192.168.217.130 dst
    =192.168.217.129 sport=22 dport=32774 use=1
```

Listing 3.2: An example *Conntrack* entry

Each entry´s signification is described in Table 3.2. The first two attributes describe which protocol the connection uses. The third attribute describes when the entry times out. If the time out value reaches 0 the connection entry is completely removed from the table. Each time a packet is seen for a given connection, the time out value is restored to the default value, which depends on protocol and state. Hence as long as there is traffic between the two network parties, the entry rests in the table. The fourth attribute tells us in what protocol specific state the connection is in. Note that *Conntrack* state names are different[8] to the ones used by the firewall (mentioned in Section 3.4.2). Attributes five to eight store the packets source and destination addresses and ports. The next attribute describes if traffic has been seen in both directions. It is this attribute which decides if a connection can be regarded as ESTABLISHED or NEW by the firewall. Entries 10 - 13 describe what the *Conntrack* module expects from a reply packet which would belong to this connection.

#	Attribute	Description
1	tcp	Used protocol (in this case TCP)
2	6	Used protocol in decimal coding
3	117	Timeout value in seconds
4	SYN_SENT	State of the connection within the *Conntrack* module
5	src=192.168.1.6	Source address of the observed packet
6	dst=192.168.1.9	Destination address of the observed packet
7	sport=32775	Source port of the observed packet
8	dport=22	Destination port of the observed packet
9	[UNREPLIED]	Keyword which specifies if return traffic was observed
10 - 13	src, dst, sport, dport	Expected source, destination addresses and ports of a reply packet belonging to this connection

Table 3.2: *Conntrack* table entry and their meaning.

In the following sections, we will have a look at the different possible *Conntrack* entries and what sort of information they gather for selected protocols.

Default Connections

This is a sort of fall back method which *Conntrack* uses if no specific module is present for the given packet's protocol. This may happen when it does not know the protocol or

[8] This is especially apparent for TCP connections, where the name of the TCP's connection state is used. These are described in Section 3.4.1.

how it functions. Basic attributes such as source and destination IP-addresses are stored and help identify a packet belonging to a connection. The first packet is regarded as NEW and the first reply packet changes the state of the connection to ESTABLISHED. From there on all packets are regarded as ESTABLISHED until the connection times out. The default time out value for *Default Connections* is 600 seconds.

UDP

UDP is a connectionless protocol. This means it does not need to guarantee that packets arrive in a certain order or at all. Therefore one would think that such a protocol would be stateless. We can, however, assign a UDP connection a state. Similar to *Default Connections*, described above, the first UDP packet would be considered as NEW. A reply packet would move the connection's state to ESTABLISHED. The following traffic would be considered to be passing an ESTABLISHED connection. Listing 3.3 shows this situation. The first entry (Line 1) flags the UDP connection as `[UNREPLIED]`. The second entry (Line 3) represents the same entry after it has seen a reply packet belonging to this connection. Note that the timeout value has changed to 180 seconds and that the `[UNREPLIED]` attribute has been removed.

```
1 udp 17 20 src=192.168.1.2 dst=192.168.1.5 sport=137 dport=1025 [
    UNREPLIED] src=192.168.1.5 dst=192.168.1.2 sport=1025 dport=137
    use=1
2
3 udp 17 180 src=192.168.1.2 dst=192.168.1.5 sport=137 dport=1025 src
    =192.168.1.5 dst=192.168.1.2 sport=1025 dport=137 [ASSURED] use=1
```

Listing 3.3: An example UDP *Conntrack* entry

To distinguish UDP entries within *Conntrack* the following attributes are considered:

- Source and Destination IP-Addresses
- Source and Destination Ports

The default time out value for UDP connections is 180 seconds.

TCP

TCP is a widely used protocol. Unlike the UDP counterpart, which can directly start sending data, TCP needs to establish a connection before sending any data. The connection establishment is done with a three way handshake [11]. The first packet is sent from the client where the SYN flag is set notifying the server that it would like to create a new connection. If the server is capable of allowing new connections, it replies to the request with a TCP packet where the SYN and ACK flags are set. The client then acknowledges the new connection with an ACK packet, and henceforth data can be exchanged. *Conntrack* follows the handshake and stores each connection state within the table as shown in Listing 3.4.

```
1 tcp 6 117 SYN_SENT src=192.168.1.2 dst=192.168.1.5 sport=1032 dport
     =80 [UNREPLIED] src=192.168.1.5 dst=192.168.1.2 sport=80 dport
     =1032 use=1
2
3 tcp 6 57 SYN_RECV src=192.168.1.2 dst=192.168.1.5 sport=1032 dport=80
      src=192.168.1.5 dst=192.168.1.2 sport=80 dport=1032 use=1
4
5 tcp 6 431999 ESTABLISHED src=192.168.1.2 dst=192.168.1.5 sport=1032
     dport=80 src=192.168.1.5 dst=192.168.1.2 sport=80 dport=1032 [
     ASSURED] use=1
```

Listing 3.4: An example TCP *Conntrack* entry observed by the client.

As already mentioned above, TCP has connection states [19]. While *Conntrack* keeps track of each state the connection is in, *IPTables* is only interested if a connection is in the NEW or ESTABLISHED state. Therefore *Conntrack* marks the connection as NEW after it has seen the first ACK flag (SYN_SENT TCP-state). After the SYN/ACK packet (SYN_RCV TCP-state) it changes the state to ESTABLISHED. Figure 3.4 shows how the client and server *Conntrack* module see the connections once a certain packet has been observed. After the client has sent the SYN packet to the server, both *Conntrack* modules mark the connection as NEW. Once the server replies with a SYN/ACK it changes is state to ESTABLISHED. The client makes the transition to the ESTABLISHED state once it has received the SYN/ACK packet.
In Listing 3.4, the NEW and ESTABLISHED states are never used explicitly, but notice how in line 1 the `[UNREPLIED]` flag is set, marking the connection as NEW, and as soon as reply traffic was observed this flag is removed (as of line 2), and hence the connection is regarded as ESTABLISHED.

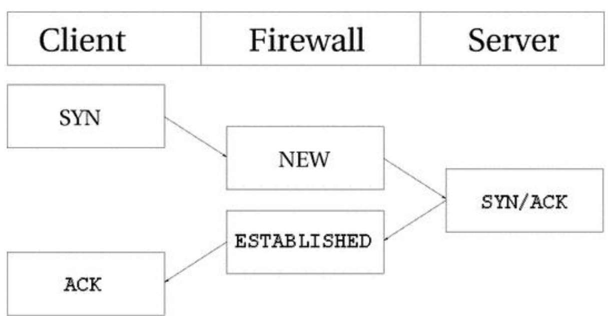


Figure 3.4: *IPTables* states during the TCP connection establishment [18].

Again opposed to UDP, where the connection can seize the transfer of data at any time, TCP needs to terminate an active connection. This is done by sending a TCP packet where the FIN/ACK flags are set to the other communication party. The other side acknowledges the connection termination with a packet where the ACK flag is set. Again *Conntrack* keeps track of the TCP-states at all times, but *IPTables* would like to now if a connection is still ESTABLISHED or not. In TCP, a connection that has terminated may still be active, this way it can still receive packets that might have been held up by congested networks between the communication parties. Therefore the connection is still considered as ESTABLISHED as can be seen in Figure 3.5.

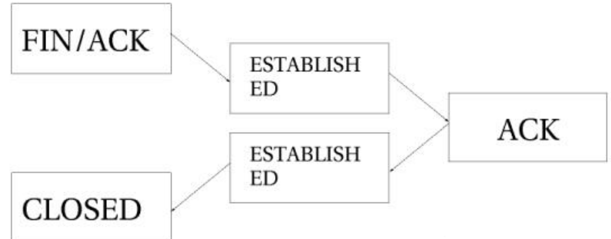


Figure 3.5: *IPTables* states during the TCP connection termination [18].

Opposed to the connection timeouts of the other communication protocols discussed above, TCP timeouts are dependent in which state the connection is in. As already mentioned, *Conntrack* keeps track of TCP-states described in [19]. Table 3.3 lists these timeout values for each TCP-state.

State	Value
NONE	1800 seconds (30 minutes)
ESTABLISHED	432000 seconds (5 days)
SYN_SENT	120 seconds
SYN_RECV	60 seconds
FIN_WAIT	120 seconds
TIME_WAIT	120 seconds
CLOSE	10 seconds
CLOSE_WAIT	43200 seconds (12 hours)
LAST_ACK	30 seconds
LISTEN	120 seconds

Table 3.3: Timeout values for each TCP state [16].

ICMP

ICMP packets are mainly used for controlling and error reporting. An ICMP connection is never regarded as ESTABLISHED since after the reply packet the stream is considered to be finished [16]. Again the source and destination IP-addresses are used to identify a connection. Other attributes are stored as well. The `type` and `code` attribute describe the ICMP control message. The `id` attribute is also used to distinguish ICMP connections. Therefore a reply message needs to carry the same `id` number in order to be identified to the same connection.

Listing 3.5 shows an example ICMP *Conntrack* entry. For the first entry of the connection the `[UNREPLIED]` flag has been set. This shows that the connection is in the NEW state. When a reply packet arrives it is considered to be ESTABLISHED. However, the ICMP reply packet also signals the end of this connection, since we know for sure that after an ICMP reply packet there is no more legal traffic. Therefore the entry is removed from *Conntrack*. The default time out value for an ICMP connection is 30 seconds.

```
1 icmp 1 29 src=192.168.217.129 dst=192.168.217.130 type=8 code=0 id
    =33072 packets=3 bytes=252 [UNREPLIED] src=192.168.217.130 dst
    =192.168.217.129 type=0 code=0 id=33072 packets=0 bytes=0 mark=0
    use=1
```

Listing 3.5: An example ICMP *Conntrack* entry

Another important part of ICMP messages is that they can be RELATED to other connections. ICMP may be used to inform a client about unsuccessful UDP or TCP connection attempts. Therefore the ICMP message should always spawn back to these connection attempts.

3.4.2 Match-State

In *IPTables*, packets can be related to tracked connections with four different states. The firewall's match-rule, which checks connection states, may be configured for the following states [16]:

- NEW: This state tells the firewall that the packet has been seen for the first time. These packets usually arise when a network component is trying to establish a new network connection.
- ESTABLISHED: This state has seen network traffic in both directions. A connection arrives in this state after the host, which sent out a packet to another host, has received a reply packet. This would imply that the firewall would change the state of the NEW connection to ESTABLISHED after receiving the reply packet. All following packets, that belong to this connection, are then considered ESTABLISHED until the connection is closed or times out.
- RELATED: A connection is considered RELATED if it belongs to an already existing connection. This connection will spawn another connection outside of its main connection. This spawned connection is then considered RELATED. An example application which uses RELATED connections is FTP. After setting up an initial connection, which is used to exchange FTP-commands, a data-stream connection is established between server and client. This second connection would be considered RELATED by the firewall. Another example of a RELATED packet would be ICMP error messages. Connection attempts between client and server using TCP or UDP may sometimes fail. In such a case ICMP is used to inform of this situation, and the packet is considered RELATED to the already existing connection attempt.
- INVALID: A packet which could not be identified or does not have a state is considered INVALID. When the *Stateful Inspection* table runs out of memory, it cannot create new entries. Therefore packets that would be considered NEW cannot be inserted into the table. Hence the packet does not have a state. ICMP messages could also be considered INVALID, if they do not respond to any known connections (RELATED).

Listing 3.6 shows a typical firewall setting for *Stateful Inspection*. All outbound traffic is allowed, whereas inbound traffic is only allowed if a connection was opened beforehand.

Lines 1-2 set the policy of the incoming and outgoing default chains (INPUT, OUTPUT) to DROP. Line three states that packets, that are outbound (OUTPUT chain), may pass if they already belong to an existing connection (ESTABLISHED) or if the packet has been seen for the first time (NEW). Line four only allows incoming (INPUT chain) packets to pass, if they actually belong to an existing connection (ESTABLISHED). Therefore in this setup an intruder from the outside could not open a new connection.

```
1 > iptables -P INPUT DROP
2 > iptables -P OUTPUT DROP
3 > iptables -A OUTPUT -m state --state NEW,ESTABLISHED -j ACCEPT
4 > iptables -A INPUT -m state --state ESTABLISHED -j ACCEPT
```

Listing 3.6: An example ICMP *Conntrack* entry

3.5 VeriNeC's Packet Filter Schema

Another part on which VeriNeC's firewall implementation leans upon, is the *Packet Filter* service of the *Network Definition* schema. At the beginning of this thesis the schema already existed and therefore was a good reference point to extract what main features the VeriNeC firewall implementation should have. This section will have a look at the most important elements defined within the *Network Definition* schema for the *Packet Filter* service, which were not already covered by *IPTables*. The complete schema can be found in Appendix B.2.

The root element for the *Packet Filter* service is called `packet-filters`. This element has three attributes called `global-out`, `global-in` and `forward`. These attributes reference a *Packet Filter* chain which will be consulted by default for incoming, outgoing or forwarded packets. The element has two child elements called `interface-filter-mappings` and `packet-filter-chain`. The `interface-filter-mappings` element is used to override the default chains defined by the `packet-filters` attributes. It has a child element called `if-map` which consists of three attributes. The first two attributes (`interface` and `direction`) state preconditions which have to be met so that the default chain is overridden for the chain defined by the third attribute (`chain`). The `interface` attribute is a reference (IDREF) to an `ethernet-binding` element. It specifies for which interface the chain is consulted. The second attribute `direction`, specifies in which direction the packet is heading. The third attribute `chain` then references the chain, which is to be consulted. More information about the `interface-filter-mappings` element can be read about in Section 4.2.3.

The `packet-filter-chain` element has two attributes called `name` and `id`. The `id` attribute identifies the chain and therefore must be non-ambiguous. This element has two child elements called `default-policy` and `packet-filter-rule`. Each chain has one default policy and consists of a set of rules. Section 4.2.3 describes the `default-policy` element's functionality in more detail.

The `packet-filter-rule` element has an `id` attribute so that the rule can be identified. It has a set of match patterns that are specified within the `packet-match-list` child element. The `packet-action-list` child element lists all possible actions a rule can execute. Only one action can be associated with a rule, with the log action element

being an exception. This element can be defined coeval with another action. All possible match-patterns specified in the `packet-match-list` element are discussed in Section 4.2.1. All possible actions described within the `packet-action-list` are described in Section 4.2.2.

Chapter 4

Implementation

4.1 Introduction

The firewall was implemented within the VeriNeC *Simulator*. To achieve this, the functionality of a firewall had to be added to the *Simulator's* routines. Moreover the *Simulator's* logging capability had to be modified, so it would log the newly triggered firewall events.

To add firewall routines to the VeriNeC *Simulator*, the already existing *Simulator* was extended. The *Simulator* was implemented using the DESMO-J framework [20] and Java 1.4. The firewall itself, however, is a module which is nearly independent of the DESMO-J framework. The only component that needs the framework is *Stateful Inspection* (see Section 4.3) which makes use of the *Simulator's* time. Another part of the *Simulator* that needed to be extended, was the event logging, which will be discussed in section 4.4.

As already mentioned in section 3.1 the VeriNeC firewall implementation leans heavily on *IPTables*. Some aspects of *IPTables* have been left out, however. Most noteworthy is *IPTables'* NAT feature which was not considered when implementing the VeriNeC *Simulator*'s firewall. Many other *IPTables'* rules and patterns were not included in this implementation since the *Network Definition* schema did not consider these. This schema is another part on which VeriNeC's firewall implementation leans upon. At the beginning of this thesis the schema already existed and therefore was a good reference point to extract the main features for VeriNeC's firewall implementation. In this chapter we will have a look at how these features were implemented. We will see that some parts of the firewall implementation look similar to *IPTables* implementation and other parts are features that were introduced by the *Network Definition* schema, which contradict to *IPTables* specification.

4.2 Packet-Filter

The *Packet-Filter* has a similar architecture as the XML-schema of the *Network Definition Document* (see B.3) for the `packet-filters` element. All packet-filtering relevant classes and packages can be found within the `verinec.netsim.firewall` package.

The *Packet-Filter* can be instantiated with the *Firewall* class. To successfully instantiate the firewall in the *Simulation*, the *Network Definition Document* must be passed to the constructor. This document holds all configuration parameters for the firewall.

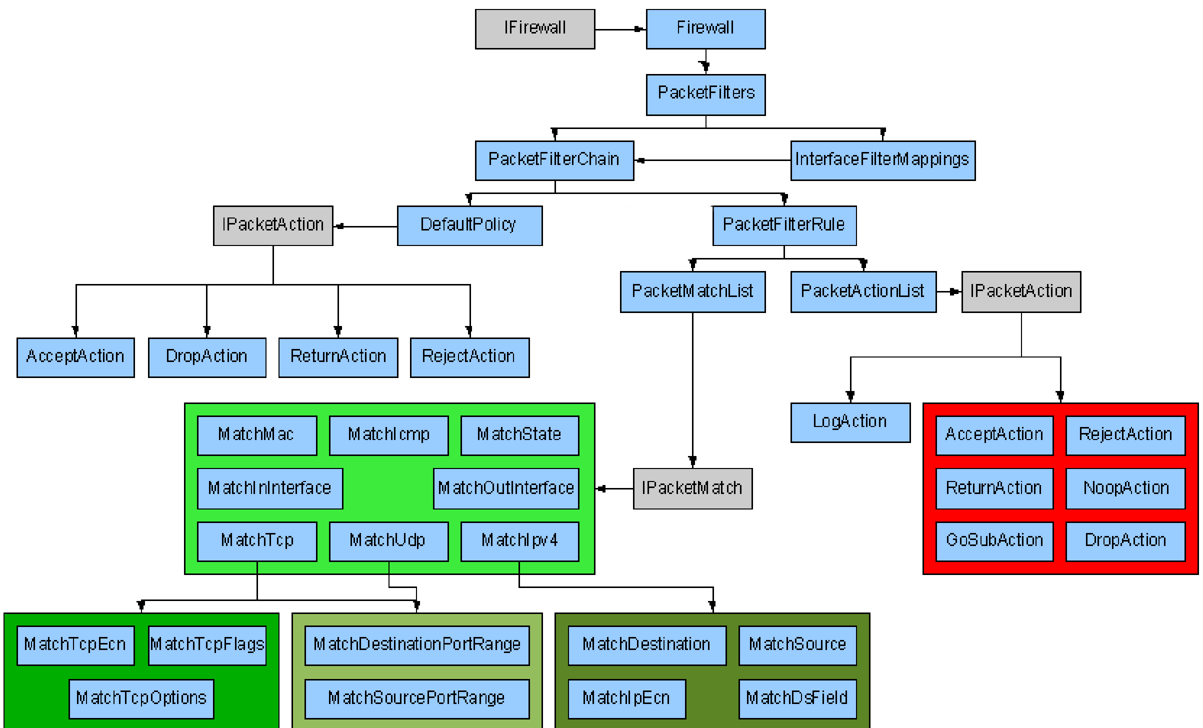


Figure 4.1: General Archiecture of VeriNeC's Firewall implementation

The *Firewall* class constructor checks if a `packet-filter` service was specified within the *Network Definition Document* and passes the child element to the *PacketFilters* constructor. Even if no `packet-filters` service was specified, the firewall constructs three default chains (INPUT, OUTPUT and FORWARD) which have the ACCEPT default policy and no rules. This way each node will have an active firewall within the simulation, even though the real-life counterpart may not have such a service running. For this occasion the *PacketFilters* class has two constructors, where the second one is used to instantiate the three default chains, when no `packet-filters` service was specified. The first one is used if the `packet-filters` service was defined within the *Network Definition Document*. The first *PacketFilters* constructor needs the *Network Definition Document* and the host name of the node which was determined by the *Firewall* class. It then checks if default chains were set within the `packet-filters` element's attributes (`global-in`, `global-out` and `forward`), and sets them accordingly. If no chain reference was specified, a default chain will be initialized, with the ACCEPT policy and containing no rules, which will later be referenced to. This way each node's *Packet-Filter* will have three default chains even if none were specified within the `packet-filters` element's attributes. The *PacketFilters* constructor then checks if a `match-state` rule was specified within any of the `packet-filter-chain` elements. If this would be the case *Stateful Inspection* is activated for this node. The constructor also sets up any `interface-filter-mappings` accordingly, which will later be referenced to if needed. The role of `interface-filter-mappings` element is described in section 4.2.3. The last action of the constructor is then to initialize each `packet-filter-chain` child element by using the *PacketFilterChain* class' constructor. The *PacketFilterChain* constructor also makes use of the *Network Definition Document* to initialize the corresponding compo-

nents, which consist of the `default-policy` element and a set of `packet-filter-rule` child elements. The available configuration options of the *DefaultPolicy* class is discussed in section 4.2.3. Each defined `packet-filter-rule` element is initialized with the *PacketFilterRule* class' constructor. This constructor reads out all rule-specific patterns which were defined in the `packet-match-list` element and sets the corresponding action defined by the `packet-action-list` element. The possible pattern-matches are discussed in section 4.2.1 while the actions are described in section 4.2.2.

The *IFirewall* interface holds all necessary methods to interact with the firewall after it has been instantiated. The most important method is `evalPacket()`, which decides if the packet is allowed to pass the firewall or not. This method will return the corresponding action (*IPacketAction*), with which the *Simulator* can call the method `excecuteAction()`. By calling this method, each returned action will then act accordingly and either send the packet on its way or block it within the simulation.

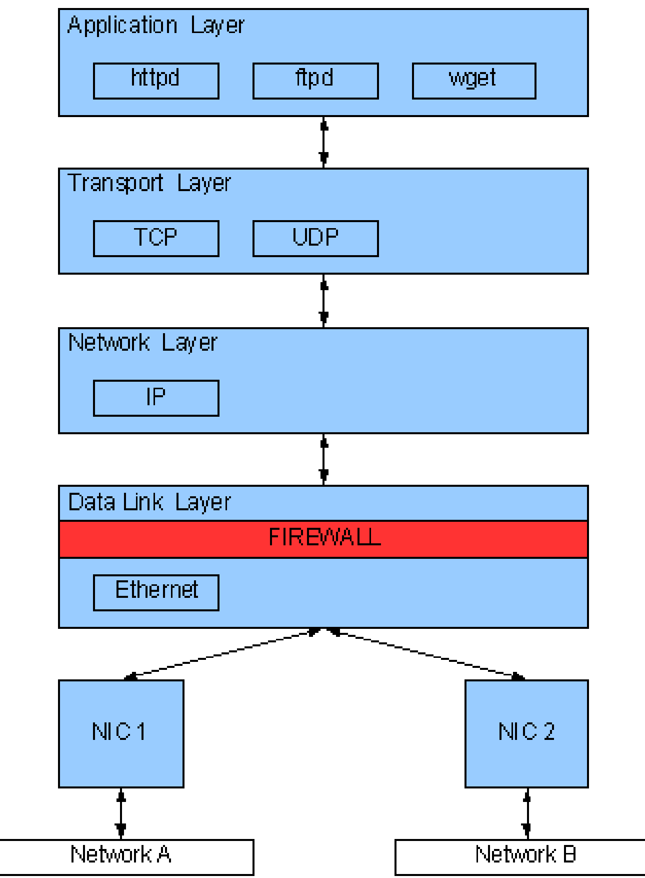


Figure 4.2: Firewall placement within the Node's network Layers

The firewall itself is nested in the *Data Link Layer* (as shown in Figure 4.2) within VeriNeC's *Simulator*. After the *Data Link Layer* has processed a packet it consults the firewall to check if it is allowed to pass or not. This is done with the `evalPacket()` method, which needs to be provided with further information. The firewall is not aware of any network activity, therefore it needs to know in which direction the packet is heading

(IN, OUT or FORWARD)[1], which NIC the packet came from or is heading to, at what simulation time the firewall is being consulted, and it needs the *Events Log* object. The latter is needed to log the firewall events to the *Simulator's* output *Log* (Figure 2.2, Section 2.2.2). Logging of firewall events is discussed in Section 4.4.

4.2.1 Rules

As already introduced in section 3.3.1, a *Packet-Filter* disposes of simple match-rules to enforce certain actions for a network packet. In VeriNeC these rules can be classified between simple and protocol specific patterns. The rules are stored in a *Vector* of *PacketFilterRule* classes within the *PacketFilterChain* class. This section will explain each match-case within VeriNeC.

Simple Rules

Simple rules define the interface through which a packet was received or being sent or what type of protocol the packet is composed of. The latter is usually followed by other protocol-specific patterns.

- In-Interface (*MatchInInterface*[2]): "Name of an interface via which a packet was received (only for packets entering the INPUT, FORWARD and PREROUTING chains). When the `!` argument is used before the interface name, the sense is inverted. If the interface name ends in a `+`, then any interface which begins with this name will match. If this option is omitted, any interface name will match" [17]. The implementation actually checks if packet is incoming, however the `+` statement is ignored[3].

- Out-Interface (*MatchOutInterface*): "Name of an interface via which a packet is going to be sent (for packets entering the FORWARD, OUTPUT and POSTROUTING chains). When the `!` argument is used before the interface name, the sense is inverted. If the interface name ends in a `+`, then any interface which begins with this name will match. If this option is omitted, any interface name will match" [17]. The implementation actually checks if the packet is outgoing, however the `+` statement is ignored[3].

- Media Access Control (MAC) (*MatchMac*): "It must be of the form XX:XX:XX:XX:XX:XX. Note that this only makes sense for packets coming from an Ethernet device and entering the PREROUTING, FORWARD or INPUT chains" [17]. The implementation actually checks if we are dealing with a correct Ethernet packet. However, since NAT was not implemented, it only checks if the packet is passing the FORWARD or INPUT chain.

- Protocol: "The specified protocol can be one of TCP (*MatchTcp*), UDP (*MatchUdp*), ICMP (*MatchIcmp*) or all. A `!` argument before the protocol inverts the test" [17]. Similar to IPTables the VeriNeC implementation of the *Packet-Filter* also

[1]Therefore the *Simulator's Data Link Layer* does the *Routing Decision* as described in Figure 3.2.

[2]Corresponding Java class name in the implementation.

[3]The VeriNeC *Network Definition* schema expects an IDREF expression, making the + statement obsolete, since the interface name needs to exist.

matches certain protocols. However, it cannot define a pattern for all protocols (all). this could be overcome by defining a pattern which checks if the packet is of type IP (*MatchIpv4*). This would be equivalent to the IPTables' statement `iptables -A INPUT -p IP ....`

- Stateful Inspection (*MatchState*): This match-case will be further discussed in section 4.3.

Similar to the `!` statement in *IPTables*, each rule can be negated with the `negate` attribute in the corresponding element. So, if for example, the `match-tcp` element's attribute `negate` would be set to true, the firewall would match to all non-TCP packets.

Protocol Specific Rules

Similar to *IPTables*, the patterns can be extended to search for protocol specific header information. In VeriNeC the IP, TCP, UDP and ICMP protocols can be matched in more detail. Here we will have a look at these protocol-specific match-cases.

MatchIpv4 An IP packet can be checked for the following attributes:

- Source Address Range (*MatchSource*): Specifies a match for an IP range of the source address. The range is specified within the `match-source` element using the `address` and `length` attributes. With these two components the range is specified in the CIDR[4] notation (address/length). For example, a CIDR notation of the IP range `192.168.1.1 - 192.168.1.255` would be `192.168.1.0/24`. The address range can be negated with the `negate` attribute. In this case an address outside of the range specified would match.

- Destination Address Range (*MatchDestination*): Same as *MatchSource* except that in this case the packet's destination address is being matched.

- Differentiated Service Code Point (DSCP) (*MatchDsField*): DSCP[5] is a field in the header of IP packets for packet classification purposes. Common IP networks have no means to distinguish packets that originate from different applications. This could cause a bottleneck for some applications (i.e. Voice over IP (VoIP)). To overcome this problem, every IP packet is prioritized by setting the 6 bits within the Type of Service byte in the IP-header. The first three bits select which class the packet belongs to and the following three bits set the drop precedence. Within VeriNeC the fields are set within the `match-dsfield` element's attributes `dscp` and `dscp-class`. The rule can be negated with `negate` attribute of the `match-dsfield` element. At the time of writing this thesis, the DSCP field was not yet implemented by VeriNeC's *Simulator*. Therefore, the pattern is ignored by the firewall, and a *Java Log*[6] entry is created, informing of this situation.

[4]Addressing scheme for the Internet which allows more efficient allocation of IP addresses than the old Class A, B, and C address scheme. The address is interpreted in a bitwise prefix-based fashion and facilitates routing by grouping address ranges block wise. `http://en.wikipedia.org/wiki/Classless_Inter-Domain_Routing`

[5]Wikipedia on DSCP `http://de.wikipedia.org/wiki/DSCP`

[6]Java Logging `http://java.sun.com/j2se/1.4.2/docs/guide/util/logging/overview.html`

- Explicit Congestion Notification (ECN) (*MatchIpEcn*): ECN is used in TCP/IP to signal the communication partner of network congestion. In the IP-header the ECN-Capable Transport (ECT) bit is set to signal the use of ECN. If the bit would be set this pattern would apply, however at the time of writing this thesis, ECN was not yet implemented by the *Simulator*. Therefore the pattern is ignored by the firewall, and a *Java Log* entry is created, informing of this situation.

MatchTcp The TCP packet can be checked for the following attributes:

- Source Port Range (*MatchSourcePortRange*): This rule matches the source port specified in a TCP packet. The lower part of the port range is defined in the `lo` attribute of the `match-source-port-range` element. While the upper level is specified with the `hi` attribute. The rule can be negated with the `negate` attribute, in which case packets, that have a source port outside of the specified range, match.
- Destination Port Range (*MatchDestinationPortRange*): Similar to MatchSource-PortRange except that the destination port is matched to the port range.
- TCP Flags (*MatchTcpFlags*): This rule matches the TCP control flags. The following flags can be checked: SYN, PSH, URG, RST, FIN and ACK. Each flag represents an attribute of the `match-tcp-flags` element. The values that each attribute can take is either 'on', 'off' or 'dontcare' (default). Therefore, similar to *IPTables* implementation, a flag pattern can be specified, which the packet needs to fulfill so that a match-case is present. The given example in Section 3.3.1 would be specified in the *Network Definition* notation as `<match-tcp-flags syn='on' rst='off' />`. This match-rule can also be inverted by setting the `negate` attribute to true.
- TCP ECN (*MatchTcpEcn*): As already mentioned before, ECN is used in the TCP/IP to signal network congestion. In TCP the Congestion Window Received (CWR) bit can be set, which signals that congestion is taking place. The other bit, ECN Echo (ECE), acknowledges the congestion. Both flags can be checked in this pattern. The rule is also invertible by setting the `negate` attribute to true. However, when this thesis was written, ECN was not yet implemented by the *Simulator*. Therefore the pattern is ignored by the firewall, and a *Java Log* entry is created, informing of this situation.
- TCP Option (*MatchTcpOption*): The TCP options field is an additional header entry, which may be used to signal further connection settings which are not already specified by the other header fields. The total length of the option field must be a multiple of a 32-bit word and is specified by the `kind` attribute of the `match-tcp-option` element. The rule checking can also be inverted by the `negate` attribute. At the time of writing this thesis, the TCP-option field was not yet implemented by the *Simulator*. Therefore the pattern is ignored by the firewall, and a *Java Log* entry is created, informing of this situation.

MatchUdp At the time of writing this documentation, the UDP protocol was not yet implemented by the VeriNeC *Simulator*. Thus, UDP rules are ignored by this implementation of the *Packet-Filter*. For completeness, however, the rules and their meaning will be explained here. UDP packets can be checked for the following attributes:

- Source Port Range (*MatchSourcePortRange*): Same as the *MatchSourcePortRange* for TCP packets introduced above, except that in this case the source port is matched for UDP packets.

- Destination Port Range (*MatchDestinationPortRange*): Same as the *MatchDestinationPortRange* for TCP packets introduced above, except that in this case the destination port is matched for UDP packets.

If an UDP match pattern element (`match-udp`) was defined within the *Network Definition*, a *Java Log* entry is created informing of this situation and the pattern is ignored.

MatchIcmp At the time of writing this thesis, the ICMP protocol was not yet implemented by the VeriNeC *Simulator*. Thus, ICMP rules are ignored by this implementation of the *Packet-Filter*. For completeness, however, the interpretation of this rule will be explained here. ICMP packets are usually used for the exchange of error and information messages. This protocol is never used by network applications (with the exception of the 'ping' program). In VeriNeC the ICMP message type can either be specified by the type name or the corresponding code. The `type` attribute of the `match-icmp` element defines the type name while the `code` attribute represents the ICMP code[7]. Further, to match any non-ICMP packets the `negate-icmp` attribute can be set to true. In addition the type name (`negate-type`) or the code (`negate-code`) can be negated as well. This way, one could match anything that is ICMP but not with a specific code or type name.

If an ICMP match pattern element (`match-icmp`) was defined within the *Network Definition*, a *Java Log* entry is created informing of this situation and the pattern is ignored.

Chain

Similar to IPTables, the rules are stored in a chain. In VeriNeC's firewall implementation each rule (*PacketFilterRule*) is stored in a Vector within the corresponding *PacketFilterChain* object. The rules themselves consist of a match-pattern (*PacketMatchList*) and an action (*PacketActionList*), which is executed if the rule applies. Note that a rule can only consist of one action, except for the LOG action. This action can always be specified in conjunction with another action.

4.2.2 Actions

Equal to *IPTables* implementation, each VeriNeC's firewall rule is associated with an action. If all the patterns match within the rule, the corresponding action is executed. VeriNeC has the following actions:

- ACCEPT: The packet passes the firewall.

- DROP: The packet is blocked.

- REJECT: The packet is blocked and an ICMP-message is returned to the sender. The ICMP message type can be defined by the `type` attribute within the `reject-action`

[7] `Wikipedia,ICMP,http://en.wikipedia.org/wiki/icmp`

element. The default type 3 states that the port is unreachable. Other possible types are: Net unreachable (0), host unreachable (1), protocol unreachable (2)[8]. Since, at the time of writing this thesis, ICMP has not yet been implemented by VeriNeC's *Simulator*, the REJECT action simply drops the packet without sending back an error message. Further a *Java Log* entry is created, informing of this situation.

- GOSUB: The packet is passed to a referenced chain for checking. The reference is specified with the `goto` attribute within the `gosub-action` element.
- RETURN: The chain does nothing and returns the packet to the calling chain.
- LOG: This action can be used to log firewall events within the firewall itself. This action has not been implemented, since all firewall events are logged to the *Simulator's Log File* anyway. If the `log-action` element was defined, a *Java Log* entry is created, informing of this situation.

In the *Network Definition* schema, each action element has a `count` attribute. This attribute is a relic from *IPTables* where the count field is used out of statistical reasons. In this implementation the `count` attribute is ignored and a *Java Log* entry is created if it was set.

4.2.3 Policies

In contrary to *IPTables* (see 3.3.3), VeriNeC can define policies for non-default chains. Further differences lie within what type of actions that can be defined. In *IPTables* only the ACCEPT and DROP actions can be used as a policy. In VeriNeC, however, one can define further actions, namely:

- ACCEPT
- DROP
- REJECT
- RETURN

ACCEPT, DROP and REJECT are terminating actions and therefore cause no problems when a default chain is consulted. The RETURN action, on the other hand, is a non-terminating action. This would cause a problem if a policy would simply RETURN, and therefore the *Simulator* would not know what to do with the packet. However, the RETURN action can be defined since one can overcome default chains using the `interface-filter-mappings` within the firewall's configuration.
The `interface-filter-mappings` element allows one user to overrule the default chains and let another chain process the packet. If no rule in this chain would apply, one could set RETURN as policy so that the default chain would be consulted for further processing of the packet. Therefore one must ensure that the default chain´s policy is not RETURN, otherwise the packet would simply drop and a *Java Log* entry mentions this unwanted situation.

[8]For a complete list of possible ICMP messages consult `http://en.wikipedia.org/wiki/ICMP`

```xml
<vn:packet-filters global-in="pfc1" global-out="pfc2">
        <vn:interface-filter-mappings>
          <vn:if-map interface="xyz" chain="pfc3" direction="in" />
        </vn:interface-filter-mappings>

        <packet-filter-chain name="default-in" id="pfc1">
          <vn:default-policy>
            <vn:drop-action/>
          </vn:default-policy>
          <vn:packet-filter-rule>
            <vn:packet-match-list>
              <vn:match-ipv4 />
            </vn:packet-match-list>
            <vn:packet-action-list>
              <vn:accept-action/>
            </vn:packet-action-list>
          </vn:packet-filter-rule>
        </vn:packet-filter-chain>

      <vn:packet-filter-chain name="default-out" id="pfc2">
        <vn:default-policy>
          <vn:drop-action/>
        </vn:default-policy>
        <vn:packet-filter-rule>
          <vn:packet-match-list>
            <vn:match-ipv4 />
          </vn:packet-match-list>
          <vn:packet-action-list>
            <vn:accept-action/>
          </vn:packet-action-list>
        </vn:packet-filter-rule>
      </vn:packet-filter-chain>

      <vn:packet-filter-chain name="overruling-chain" id="pfc3">
        <vn:default-policy>
          <vn:return-action/>
        </vn:default-policy>
        <vn:packet-filter-rule>
          <vn:packet-match-list>
            <vn:match-udp/>
          </vn:packet-match-list>
          <vn:packet-action-list>
            <vn:drop-action/>
          </vn:packet-action-list>
        </vn:packet-filter-rule>
      </vn:packet-filter-chain>
</vn:packet-filters>
```

Listing 4.1: Interface-Filter-Mapping example, which overrules a default chain

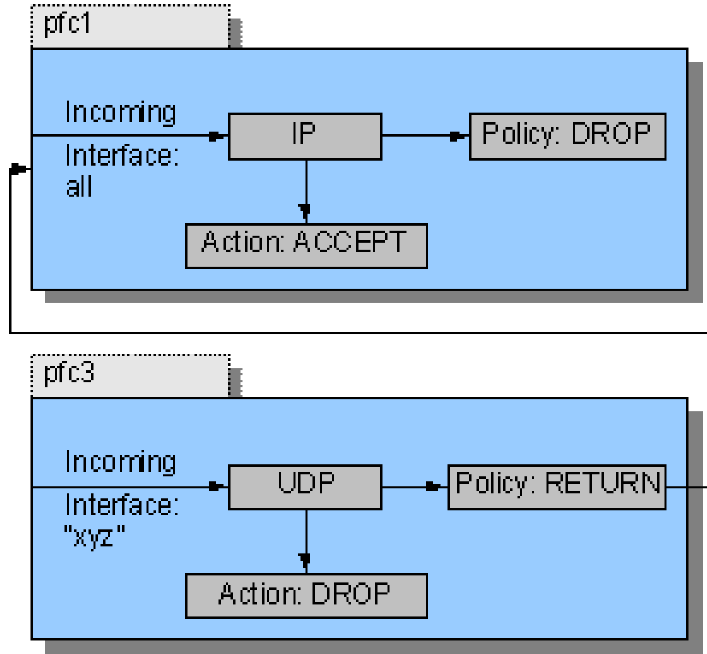


Figure 4.3: Packets traversing the example *Packet-Filter* from Listing 4.1

An example provided in Listing 4.1 demonstrates this situation. Two default chains were defined for all incoming (pfc1) and outgoing (pfc2) traffic regardless through which interface the packets were received. They both accept all IP packets and drop any other sort of traffic. However, the `interface-filter-mappings` element (Listing 4.1, lines 3-5) overrules the incoming default chain 'pfc1', if the packet is incoming over the 'xyz' interface, and uses 'pfc3' instead. In this case, if the packet is of type UDP, it is dropped and the *Packet-Filter* seizes consulting chains for this packet. With all other packets, the RETURN action would get triggered, which would induce the *Packet-Filter* to consult the default chain 'pfc1' for further processing as illustrated in Figure 4.3.

4.3 Stateful Inspection

In Section 3.4, *Stateful Inspection* was shortly introduced. It was decided that the firewall for the *Simulator* should also incorporate *Stateful Inspection* since most firewalls and routers have such a system running. However, when searching for any good documentation or system specification, I ran into a few problems. *Stateful Inspection* is not a defined standard, and it was quit hard to find any good documentation about it. To make things worse, most vendors implement *Stateful Inspection* in their own manner. This could cause problems when running the simulation for a setting where the "'to be configured"' firewall component would not react in the same manner as in the VeriNeC *Simulation*. Since it was hard to find any vendor specific implementation of sateful inspection, it was chosen to lean the implementation toward the one used in *IPTables*. All *Stateful Inspection* relevant classes and packages can be found within the `verinec.netsim.firewall.util.table` package.

As already mentioned in Section 3.4.1, Unix based operating systems have *IPTables*

run in conjunction with a kernel module called *Conntrack*. This module keeps track of the states for all active network connections. With this module *IPTables* may then make rule-based decision on dropping or letting packets through the firewall based upon these connection states. This section will explain how *Stateful Inspection* was realized for the *Simulator*. Furthermore, it will go into what was implemented and what shortcomings are present and how the *Stateful Inspection* part of the firewall could be extended.

4.3.1 The State Table

The *Conntrack* module stores the connection information in a table. It was therefore chosen, for VeriNeC's firewall implementation, to also store certain connection relevant information within a table. For this a *Java Hash Table*[9] class was used. A *Hash Table* is a construct which associates keys with values (*Records*). In Java the keys and their associated entries are objects. The comparison of keys is done with their hash value. This hash value is calculated by transforming the key object into a hash value using a hash function (`hashCode()`). The hash value then represents the index of the table (*Bucket*) where the entry object is being stored. Depending on how the hash function calculates index values, collision may occur. Collision describes the event of two keys which generate the same hash value, and therefore two *Records* would reside in the same *Bucket* creating a linked list. Figure 4.4 shows such a situation. The first *Bucket* stores two *Records* since both keys generated the same hash value. If this is the case, another method (`equals()`) is consulted to distinguish the keys. This method actually compares the object's values with each other and checks if both objects are an instance of the same class.

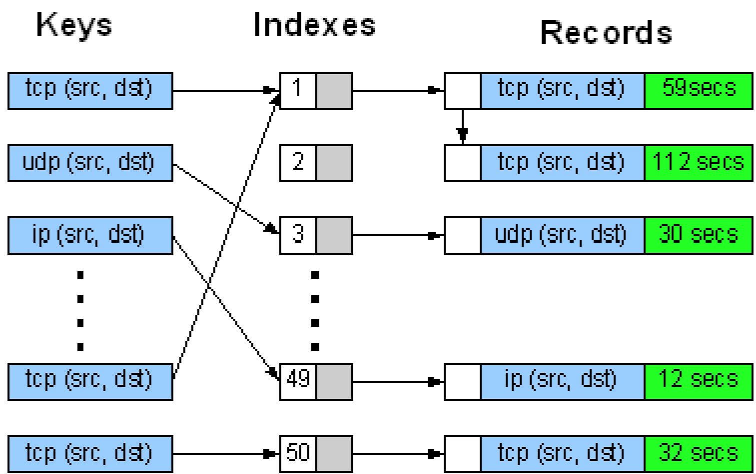


Figure 4.4: *Hash Table* example, showing two *Records* in one *Bucket* (Index 1).

The biggest advantage of *Hash Tables* is their lookup time. They normally offer a constant lookup time O(1) and in the worst case a lookup time of O(n). Therefore the tables are mostly used when large amounts of records are being stored[10].

[9]Class Hashtable, `http://java.sun.com/j2se/1.4.2/docs/api/java/util/Hashtable.html`
[10]Wikipedia, Hash Table, `http://en.wikipedia.org/wiki/Hash_table`

Key

In VeriNeC's firewall implementation the keys object represent the observed protocol and consists of selected header information of the protocol's packet (see Table 4.1). Depending on the protocol observed, the following key is used with the corresponding header information:

- IP (*IPKey*): This key is used for *Default Connections* (see Section 3.4.1). This is the case if the *Transport Layer* protocol is not recognized. The source and destination addresses are stored. The hash value is calculated (`hashCode()`) with the protocols name (IP), the source address and the destination address. The `equals()` method compares the source and destination addresses and checks that both objects are an instance of the same class. Note that all following keys use this method to distinguish themselves.
- TCP (*TCPKey*): Apart from storing the source and destination IP-addresses, the TCP key also stores source and destination ports to distinguish table entries. The `hashCode()` method uses these four components to calculate the hash value.
- UDP (UDPKey): Similar to *TCPKey* above. This key object has already been integrated into *Stateful Inspection* and therefore would be ready once the VeriNeC *Simulator* implements UDP.

Protocol Name	**Source Address**	**Destination Address**	**Port**	**Port**
'ip'	IP-Address	IP-Address	NA	NA
'tcp'	IP-Address	IP-Address	Source Port	Destination Port
'udp'	IP-Address	IP-Address	Source Port	Destination Port

Table 4.1: Content of protocol specific keys

Entry

The entry object keeps track of the timeout value and the state in which the connection resides in. For each protocol it keeps track of different timeout values. Table 4.2 lists the timeout values for each state of the connection. Notice that the timeout values for TCP entries rely in which state the TCP connection is in.

4.3.2 Workflow

Within the VeriNeC *Simulator*, *Stateful Inspection* works as follows:

- If a `match-state` rule was defined in a node's firewall configuration, *Stateful Inspection* is activated during the simulation for that node. Otherwise *Stateful Inspection* does not log any network traffic.
- If a NEW state was defined within a `match-state` element, the protocol specific header information is used to create a key object. The implementation then verifies if a connection already exists by checking if this key object is already residing in the

Connection Type	State	Timeout (Default)
Default Connection (*IPKey*)	NEW, ESTABLISHED, RELATED	6000
UDP Connection (*UDPKey*)	NEW, ESTABLISHED, RELATED	1800
TCP Connection (*TCPKey*)	NONE	18000
	ESTABLISHED	4320000
	SYN SENT	1200
	SYN RECV	600
	FIN WAIT	1200
	TIME WAIT	1200
	CLOSED	18000
	CLOSE	100
	CLOSE WAIT	432000
	LAST ACK	300
	LISTEN	1200

Table 4.2: Content of the entry objects

Hash Table. If this is the case the packet can not be in the NEW state. Otherwise the packet is considered as NEW. For TCP packets, the SYN flag also needs to be set so that the packet is considered to be in the NEW state.

- When the ESTABLISHED state was declared, the implementation checks whether the packet already belongs to an existing table entry. It does this by generating a key, corresponding to the observed packet, with which it checks if a *Record* already exists within the table. If an entry exists then the packet is considered to be in the ESTABLISHED state. A connection moves to the ESTABLISHED state as soon as it has seen a reply packet. For TCP packets, however, the first reply packet needs the SYN and ACK flags set (part of the three-way handshake) so that the connection moves to the ESTABLISHED state.

- For the INVALID state, the implementation checks if the *Hash Table* is full. If this is be the case, the packet is INVALID if it does not already belong to an existing connection. A packet would also be considered INVALID if it does not contain an IP-header.

- The RELATED state is not functional in this implementation of the *Packet-Filter*. The rule is ignored and a *Java Log* entry is created informing of this situation.

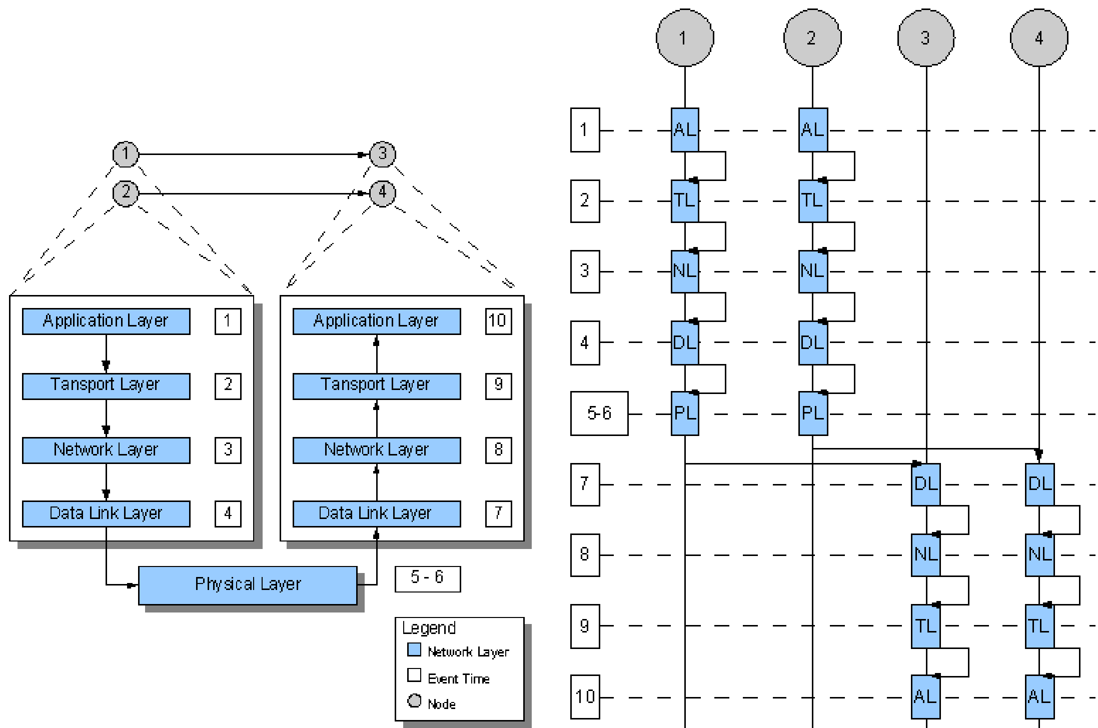
- After a packet has been accepted by a chain, depending on which state the connection is in, the corresponding entry is created or updated. This is in contradiction to *IPTables* implementation of *Stateful Inspection*, where *Conntrack* would update its table after a packet has traversed in the PREROUTING or the OUTPUT chain (see Section 3.4.1) only.

Similar to *Conntrack's* modules, the implementation stores relevant information of each protocol and handles each protocol in a different manner. This is especially apparent for TCP. As already described in Section 3.4.1, a TCP connection is established with the 3-way handshake. Therefore VeriNeC's firewall implementation considers this procedure

and actually checks if the correct flags have been set. A TCP packet is only considered to be in the NEW state when the SYN flag is set. The reply packet needs the SYN and ACK flag set, to be considered as ESTABLISHED.
The firewall's state table also stores the actual state of the TCP connection. As can be seen in Table 4.2, each TCP state holds a different timeout value, therefore it is essential that *Stateful Inspection* knows in which state the connection is in. This is done by checking the state within the used socket[11] (*FSMSocketImpl* class), which is stored in the *Transport Layer* of each node.

4.3.3 Timeout

The notion of time is not given within a DESMO-J framework since it is based on the discrete event simulation paradigm. In such a setting the simulator's 'time' simply increments each time an event was observed. Hence, time does not pass in the traditional meaning, instead it passes when something happens.



(a) Packets traversing each Node's Network Layer scheduled at a specific Event time.

(b) Sequence Diagram of the four Nodes.

Figure 4.5: Example of two Nodes sending a packet simultaneously.

Figure 4.5 shows an example of how the simulation time increments during a run. Nodes 1-2 are sending a network packet to nodes 3-4 at the same time. For each node the packet is passed from *Network Layer* to *Network Layer*, where the needed header information is added to the packet. After each *Network Layer* has processed the packet it schedules the *Simulator* to handle the packet in the next layer at the next event time. Therefore the event time increases each time the packet passes a layer within a node.

[11]The *Simulator* uses Java sockets to interface to network applications [3]

Notice, however, that each of the first two nodes handle a packet within a layer at the same event time. The packets of each node is processed at the *Application Layer* at simulation time 1. The next layer is then scheduled to process the packet at simulation time 2 and so forth. The *Physical Layer* simulates a delay which occurs in the physical network connection. Therefore it schedules the receiving ends *Data Link Layer* to process the incoming packet with a delay of 2, at simulator time 7.

At this rate, a round trip time for a request-reply packet observed by the firewall[12] would be 13. Based upon this fact, the timeout values for the *Stateful Inspection* entries had to be adjusted to respect this. A round trip time of a request-reply packet within a real network is measured in milliseconds and the timeout values within *Conntrack* are defined in seconds. Therefore it was decided to base VeriNeC´s timeout values upon *Conntrack´s* multiplied by the factor 10. Hence, the assumption is made, that the average latency within a real network would be 130 milliseconds.
The default timeout values for each protocol and state can be found in Table 4.2.

Configuration Since the default timeout values were defined out of an assumption, it is possible to alter their values within a *Java Properties*[13] file. The file is stored as XML. Each protocol's timeout value is represented as an element. To alter the timeout value, one can modify the `timeout` attribute of the corresponding element. The properties file is called `firewall_config.xml` and can be found in the `data` directory of the current Java working path[14]. The values stored within this file are valid for all nodes, thus it is not possible to alter the values for one node only.
Note that if the `firewall_config.xml` file is malformed or corrupt the default values are taken and the file is rewritten. When this happens all previously saved values are lost!
An example properties file can be found in Appendix C.1.

The default values are coded within the class of the respective protocol. So for instance, to alter the default timeout value for UDP, one would have to change the static variable `UDP_TIMEOUT` within the *UDPKey* class.

4.3.4 Extending

It was decided to make *Stateful Inspection* as expandable as possible. The key used for a *Hash Table* entry is based on an extendable architecture as can be seen in Figure 4.6. *Stateful Inspection* can be extended by adding specific inspection methods for selected protocols. The following steps can be taken to extend *Stateful Inspection´s* functionality.

- Extend *Stateful Inspection* by protocol.
- The protocol must be a member of the *Transport Layer* or above.
- Each newly created protocol class must extend the *SimpleKey* class. The *SimpleKey* class implements the *ITableKey* interface and therefore the following methods would

[12] Remember that the firewall was placed within the *Data Link Layer* as described in Section 4.2
[13] Java Properties, `http://java.sun.com/j2se/1.4.2/docs/api/java/util/Properties.html`
[14] This is valid for the current VeriNeC implementation.

need to be implemented: `handleNew()`, `handleEstablished()`, `handleRelated()` and `updateEntry()`.

- The *SimpleKey* class already implements the `equals()` and the `hashCode()` method, which are used for the *Hashtable* lookup. It is recommended however to override these methods with your own implementation.

- In order to be able to modify the timeout values found within the the *Java Properties* file, a `static` method called `initTimouts()` has to be created and implemented. The methods found in the *TCPKey* or *UDPKey* classes may serve as an example. Finally, the call to this method has to be added in the static method `loadTimeouts()` found in the *SimpleKey* class.

- To finally add the new functionality to stateful inspection, the `createKey` method within the `StatefulInspection` class needs to be modified.

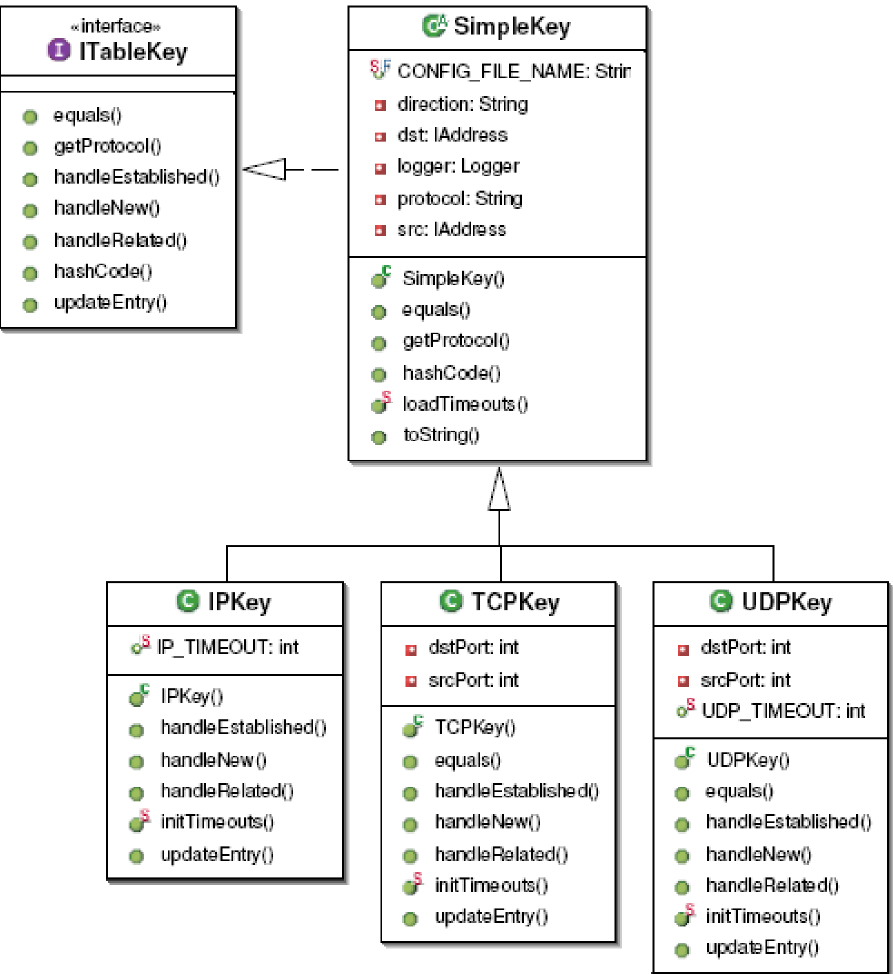


Figure 4.6: Architecture of *Stateful Inspection´s* Keys

4.3.5 Shortcomings

The current implementation of *Stateful Inspection* for the VeriNeC firewall has some shortcomings. First and foremost the RELATED match case is not yet functional. This state would be used in conjunction with ICMP or FTP protocols. The match case would take care of related traffic to be able to pass the firewall. It was chosen not to implement this match-case as of yet, since these protocols have not yet been implemented by the VeriNeC *Simulator*.
Further, it is not clear whether the timeout values chosen above are accurate enough to correctly model a network, where *Stateful Inspection* firewalls are active. The assumption taken above, should however be considered as an introductory value that can be modified as more complex simulation runs are executed and experience is collected. Further, the properties file allows for a quick modification of these values.
Thirdly not all protocols have been included in this implementation of *Stateful Inspection*. The TCP and UDP protocols have been implemented. Other protocols however are processed with the fall back method *Default Connections* which tracks IP connections. More protocols can be added using the method described in Section 4.3.4.

4.4 Logging

Within the simulation, the firewall events are logged to the *Simulator's Log Document*, similar to other events that were triggered by the *Input File* (see Figure 2.3). Since the *Simulator* was extended with a firewall, the *Log File* had to be adapted. This was accomplished by adding firewall specific events to the events schema file. The added specification can be seen in Appendix B.4.

```
1  <event time="2" node="node" layer="2" service="packet-filters" id="75
       e4fc:10c80b9024c:-7ff3">
2   <firewall type="triggered-action" action="accept" chainID="pfc4">
3    <packet-filter-rule>
4     <match-ipv4>
5      <match-source negate="false">
6       <iprange ip="192.168.0.0" length="16" />
7      </match-source>
8     </match-ipv4>
9    </packet-filter-rule>
10  </firewall>
11 </event>
12 ...
13 <event time="5" node="node" layer="2" service="packet-filters" id="75
       e4fc:10caf5cf5ce:-7ff0">
14  <firewall type="default-policy-action" action="drop" chainID="pfc1"
        />
15 </event>
```

Listing 4.2: Example firewall entries in the *Simulator's Log File*

Two sorts of firewall events are logged during a simulation run. The `triggered-action` and the `default-policy-action`. Both elements describe which action (`action` at-

tribute) was executed and which chain (`chainID` attribute) triggered the action. Rule details follow the log-entry, if the action was triggered by a matching rule. The `action` attribute specifies which action was taken. If a `triggered-action` was logged then the action is associated with the logged rule, and therefore was defined within the rule's `packet-action-list` element. For a `default-policy-action`, the action is associated with the one defined within the chain's `default-policy` element. Note, if no `chainID` was specified, then the action was executed by a default chain, which was not defined within the node's `packet-filters` element. As described in Section 4.2, these chains are initialized if no default INPUT, OUTPUT or FORWARD chains were specified within the mentioned `packet-filters` element.

Listing 4.2 shows two example firewall-log entries. The first log entry (lines 1-11) was triggered by a matching rule (the source IP-address matched to the range specified by the rule), therefore the rule details are logged as well. The second entry (lines 13-15), however, executed a chain policy, due to the fact that none of the rules applied.

Chapter 5

Conclusion

The VeriNeC *Simulator's* firewall has been implemented in such a way, that it correctly depicts a real world counterpart firewall in a network. It does basic packet-filtering based upon the packet's header information. Further the notion of *Stateful Inspection* was implemented, which simulates the tracking of network connection states with which firewall decisions can be based upon.

Stateful Inspection

Stateful Inspection was implemented based upon *IPTables*. The main drawback of this proceeding lies within the notion of timeouts. *IPTables* timeouts are based upon time, whereas within the simulation the notion of time increments when events occur. Therefore the timeout values chosen for the simulation have to be taken with a grain of salt, as it is not clear if the set timeout values actually can simulate a correct run for *Stateful Inspection*. Within time and further tests, however, these values can be perfected and set within the firewall´s *Stateful Inspection* configuration file (see Section 4.3.3)
Default Connections (IP connection) and TCP connection tracking was implemented for *Stateful Inspection* (see Section 4.3.1). Further, UDP connection tracking was also implemented even though the *Simulator* does not support this protocol as of yet. If further protocol specific connection tracking would need to be implemented, this would be possible as described in Section 4.3.4. The NEW, ESTABLISHED and INVALID states can be checked in this firewall implementation, the RELATED state, however, has not yet been implemented. Thereto, protocols that could be RELATED to a connection would need to be implemented within the *Simulator* (i.E. ICMP or UDP). FTP uses RELATED connections since two TCP connections are used for an FTP transaction. But, since there was no way to create a FTP setting within the *Simulator* to test the correct functionality of *Stateful Inspection*, this feature was dropped.

NAT

This version of the firewall implementation comes without any NAT capabilities. It would be a nice extra feature if at one point the firewall would be extended to consider such a component. Thereto the *Network Definition* schema needs to be updated so that NAT would be configurable, and the firewall would need to be extended so that the NAT rules would be checked.

It would also be possible to introduce NAT separately from the firewall implementation. As already mentioned in Section 4.2 the routing decision would need to be undertaken by the *Data Link Layer*. Therefore this layer could consult an independent implementation of NAT before the firewall is consulted for incoming traffic, and for outgoing traffic the firewall would be consulted before NAT comes into play (as described in Figure 3.2).

Future Work

The firewall's functionality is based upon the elements, that were defined within the *Network Definition* XML schema (node.xsd). Some of the defined elements could not be implemented, however. This is due to the fact that some aspects of network simulation has not yet been implemented by VeriNeC's *Simulator*. Once these features would be added, the following `match()` method (which correspond to the added protocol / feature) need to be implemented within these classes:

- *MatchDsField* (see 4.2.1 DSCP)
- *MatchIpEcn* (see 4.2.1 ECN for IP)
- *MatchTcpEcn* (see 4.2.1 ECN for TCP)
- *MatchTcpOption* (see 4.2.1 TCP Option)
- *MatchUdp* (see 4.2.1 UDP)
- *MatchIcmp* (see 4.2.1 ICMP)

Subjective View of the Project

Generally, good experience was collected during the time of this project. I would like to thank Dominik Jungo and David Buchmann, who always had time and an open ear for questions or suggestions, or helping me out when I was stuck. I think in time, the VeriNeC project can yield into a very powerful tool for network administrators, which are assigned to administer large heterogeneous network setups.

Bibliography

[1] Dominik Jungo, David Buchmann and Ulrich Ultes-Nitsche: *Verified Network Configuration Project*, http://diuf.unifr.ch/tns/projects/verinec/

[2] Dominik Jungo, David Buchmann and Ulrich Ultes-Nitsche: *The Role of Simulation in a Network Configuration Engineering Approach*, Proceedings of the International Conference ICICT on Multimedia Services and underlying Network Infrastructure. ICICT 2004, Cairo, Egypt, December 2004., http://diuf.unifr.ch/tns/projects/verinec/reports.html

[3] Dominik Jungo, David Buchmann and Ulrich Ultes-Nitsche: *A Unit Testing Framework for Network Configurations*, University of Fribourg, Switzerland

[4] Dominik Jungo, David Buchmann and Ulrich Ultes-Nitsche: *Classification of network configuration rules in VeriNeC*, University of Fribourg, Switzerland

[5] Dominik Jungo, David Buchmann and Ulrich Ultes-Nitsche: *Testing of semantic properties in XML documents*, University of Fribourg, Switzerland

[6] David Buchmann: *Verinec Translation Module*, University of Fribourg, Switzerland

[7] David Buchmann: *Automated Configuration Distribution in VeriNeC*, ICETE 2005, Reading, UK October 3-7, 2005., http://diuf.unifr.ch/people/buchmand/2005_ICETE_Verinec.pdf

[8] Patrick Aebischer (2005): *Network Sniffer, Ein Modul für Verinec*, University of Fribourg, Switzerland

[9] Geraldine Antener (2005): *ConfigImporter, A new component for the VeriNeC project*, University of Fribourg, Switzerland

[10] Nadine Zurkinden (2005): *WindowsResearch, Implementation of a Translator for Windows Machines within the Project VeriNeC*, University of Fribourg, Switzerland

[11] Andrew S. Tanenbaum (2003): *Computer Networks*, Forth Edition, Prentice Hall

[12] C. Pfleeger and S. Pfleeger (2003): *Security in Computing*, 3rd Edition, Prentice Hall, ISBN: 0130355488

[13] Andreas Lessig (2003): *Linux-Firewalls - Ein praktischer Einstieg*, O'Reilly, ISBN: 3-89721-357-5, http://www.oreilly.de/german/freebooks/linuxfireger/

[14] Norbert Pohlmann (2001): *Firewall-Systeme - Sicherheit für Internet und Intranet*, MITP-Verlag Bonn, ISBN: 3-8266-0719-9

[15] Stefan Strobel (1999): *Firewalls - Einführung, Praxis, Produkte*, dpunkt.verlag Heidelberg, ISBN: 3-932588-49-5

[16] Oskar Andreasson (2005): *Iptables Tutorial 1.2.0*, `http://iptables-tutorial.frozentux.net/iptables-tutorial.html`

[17] Herve Eychenne [et al.] (2002): *Manpage of Iptables*, `http://www.manpage.org/cgi-bin/man/man2html?query=iptables`

[18] Rusty Russell (2002): *Linux 2.4 Packet Filtering HOWTO*, `http://www.netfilter.org/documentation/HOWTO/packet-filtering-HOWTO.html`

[19] Gordon McKinney (2004): *TCP/IP State Transition Diagram (RFC793)* `http://gmckinney.info/resources/TCPIP_State_Transition_Diagram.pdf`

[20] *Discrete Event Simulation and Modeling in Java*, `http://www.desmoj.de`

Appendix A

Acronyms

VeriNeC Verified Network Configuration

DESMO-J Discrete-Event Modeling and Simulation in Java

API Application Programming Interface

XML eXtensible Markup Language

NIC Network Interface Card

DNS Domain Name System

NAT Network Address Translation

HTTP HyperText Transport Protocol

FTP File Transfer Protocol

DHCP Dynamic Host Configuration Protocol

IP Internet Protocol

TCP Transport Control Protocol

UDP User Datagram Protocol

ICMP Internet Control Message Protocol

MAC Media Access Control

LAN Local Area Network

IDS Intrusion-Detection System

DSCP Differentiated Service Code Point

ECN Explicit Congestion Notification

ECT ECN-Capable Transport

CWR Congestion Window Received

ECE ECN Echo

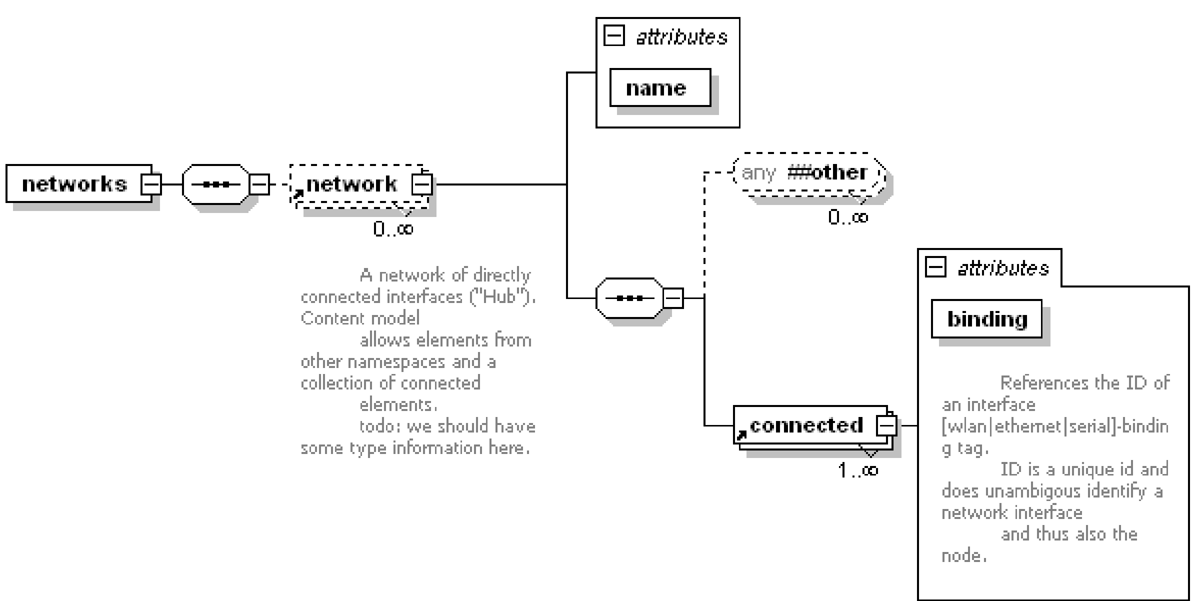
CIDR Classless Inter-Domain Routing

VoIP Voice over IP

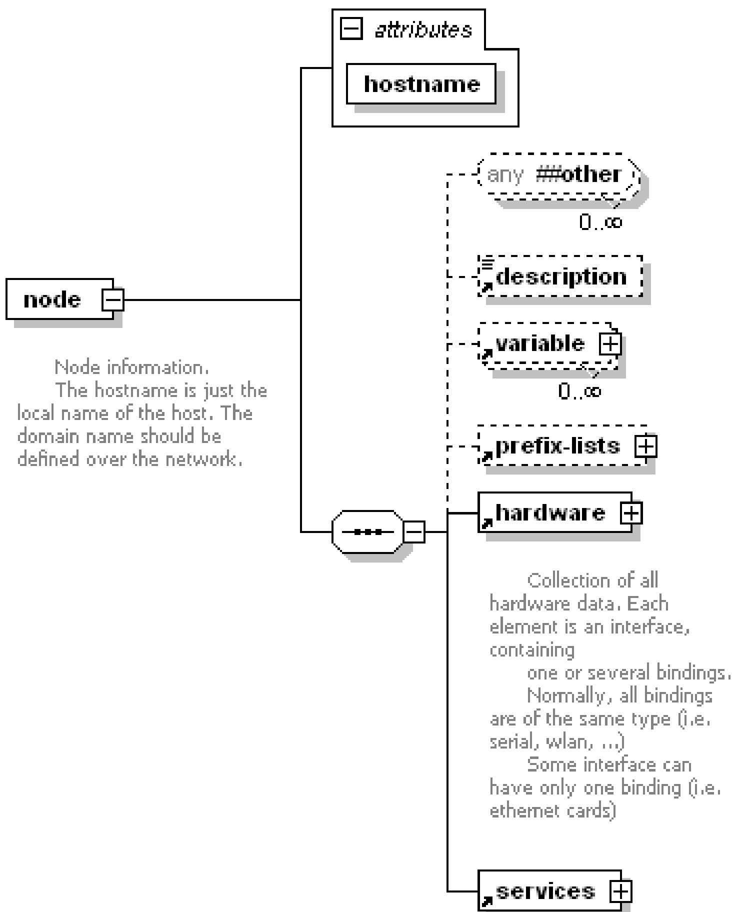
Appendix B

Verinec Schemas

B.1 Network Topology Schema from network.xsd

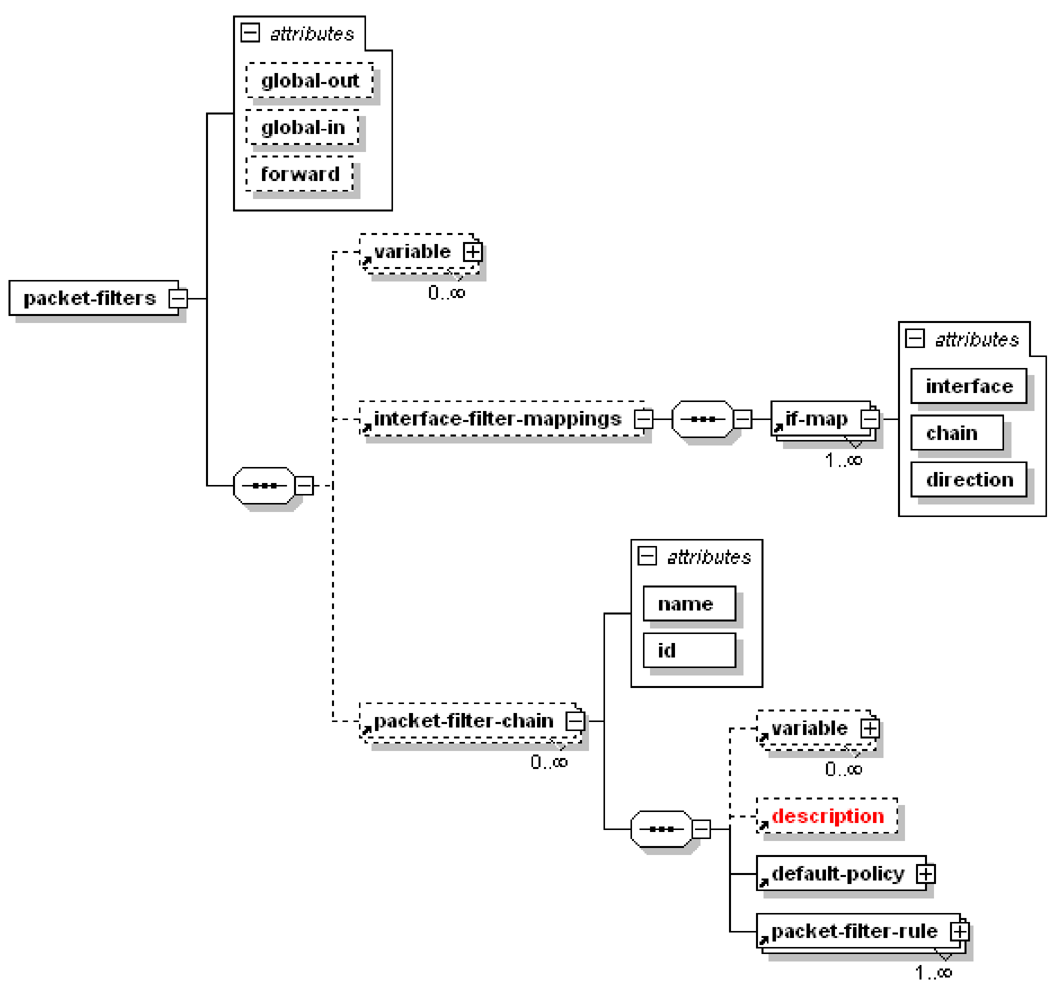


B.2 Node Schema from node.xsd

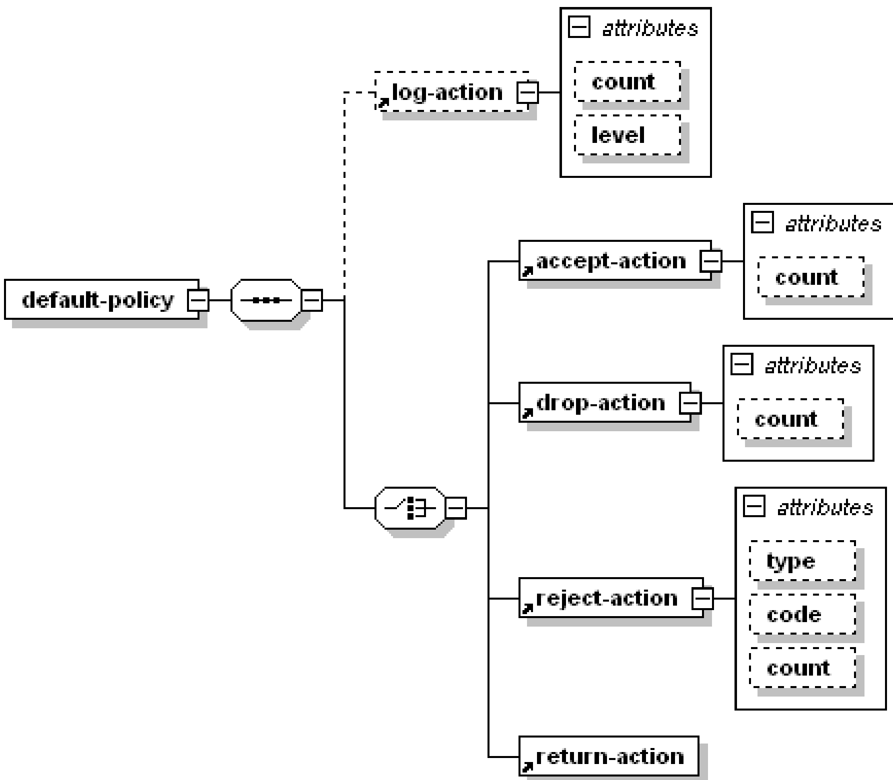


B.3 Packet-Filter Schema from node.xsd

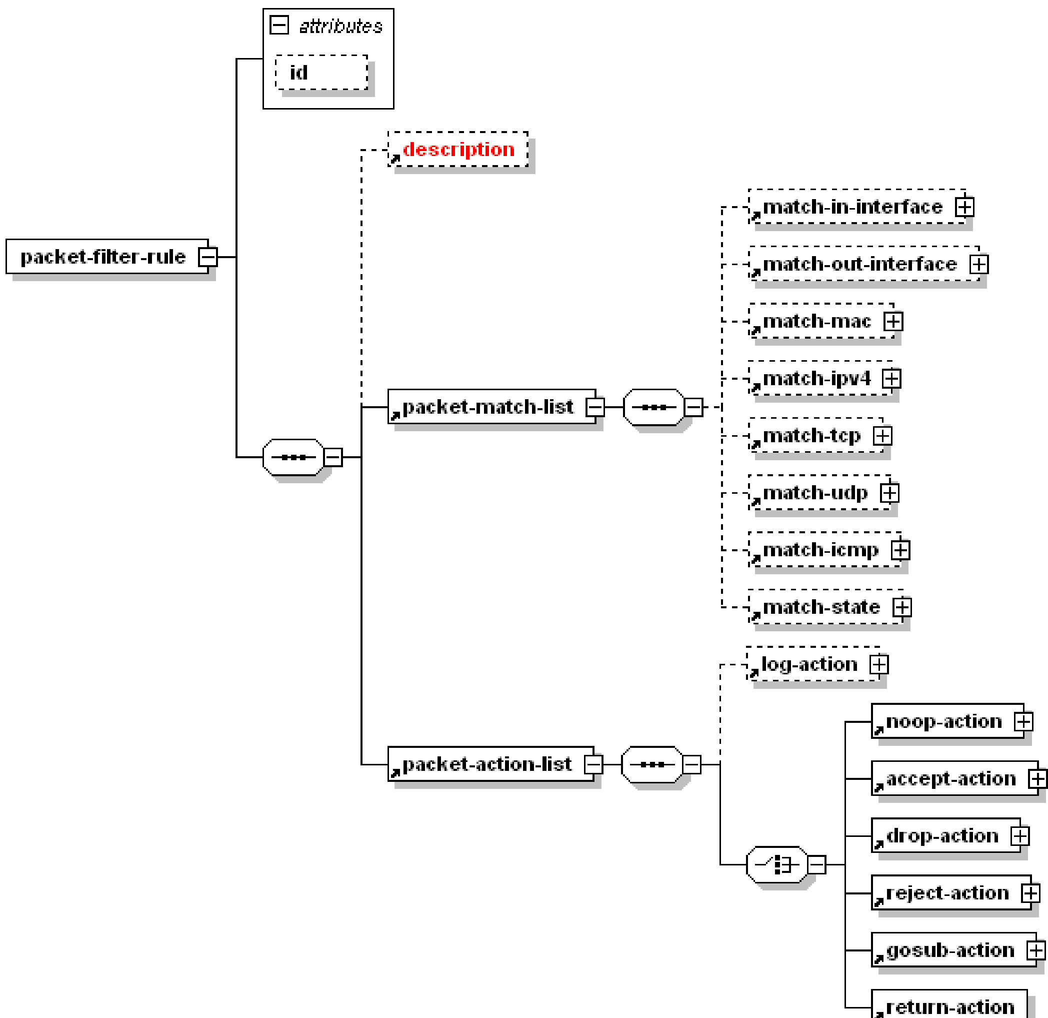
B.3.1 packet-filters



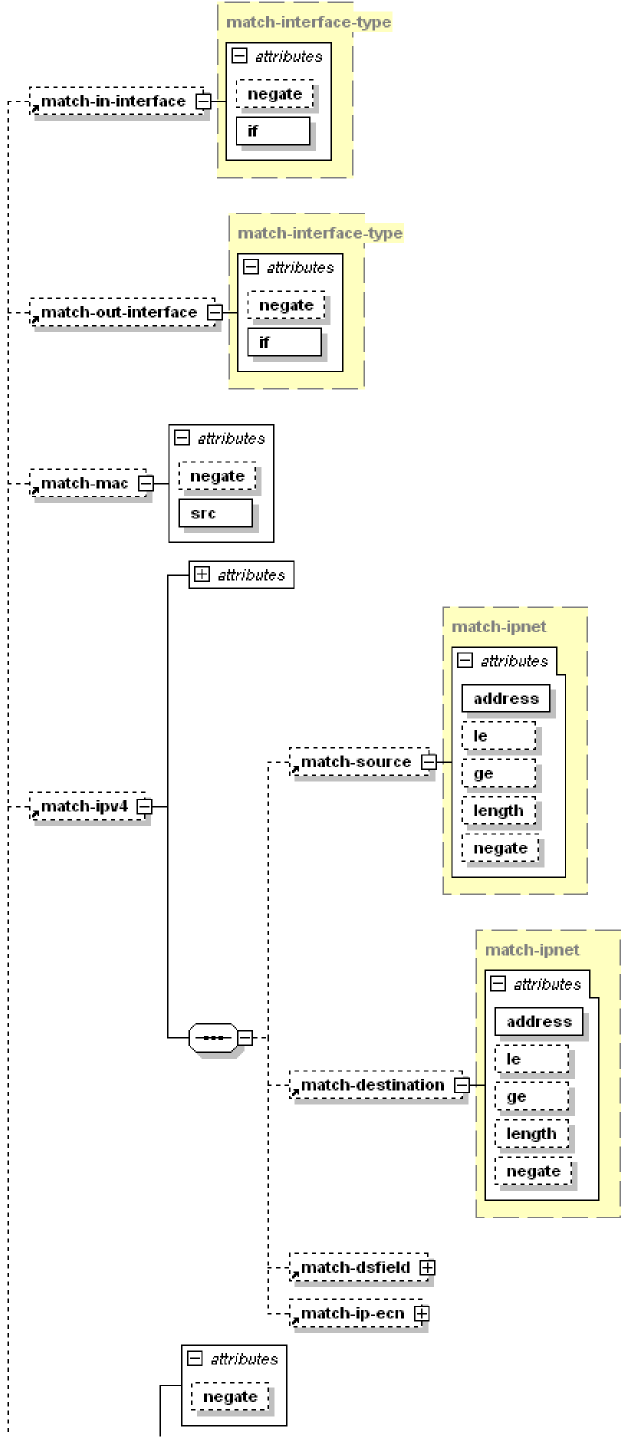
B.3.2 default-policy

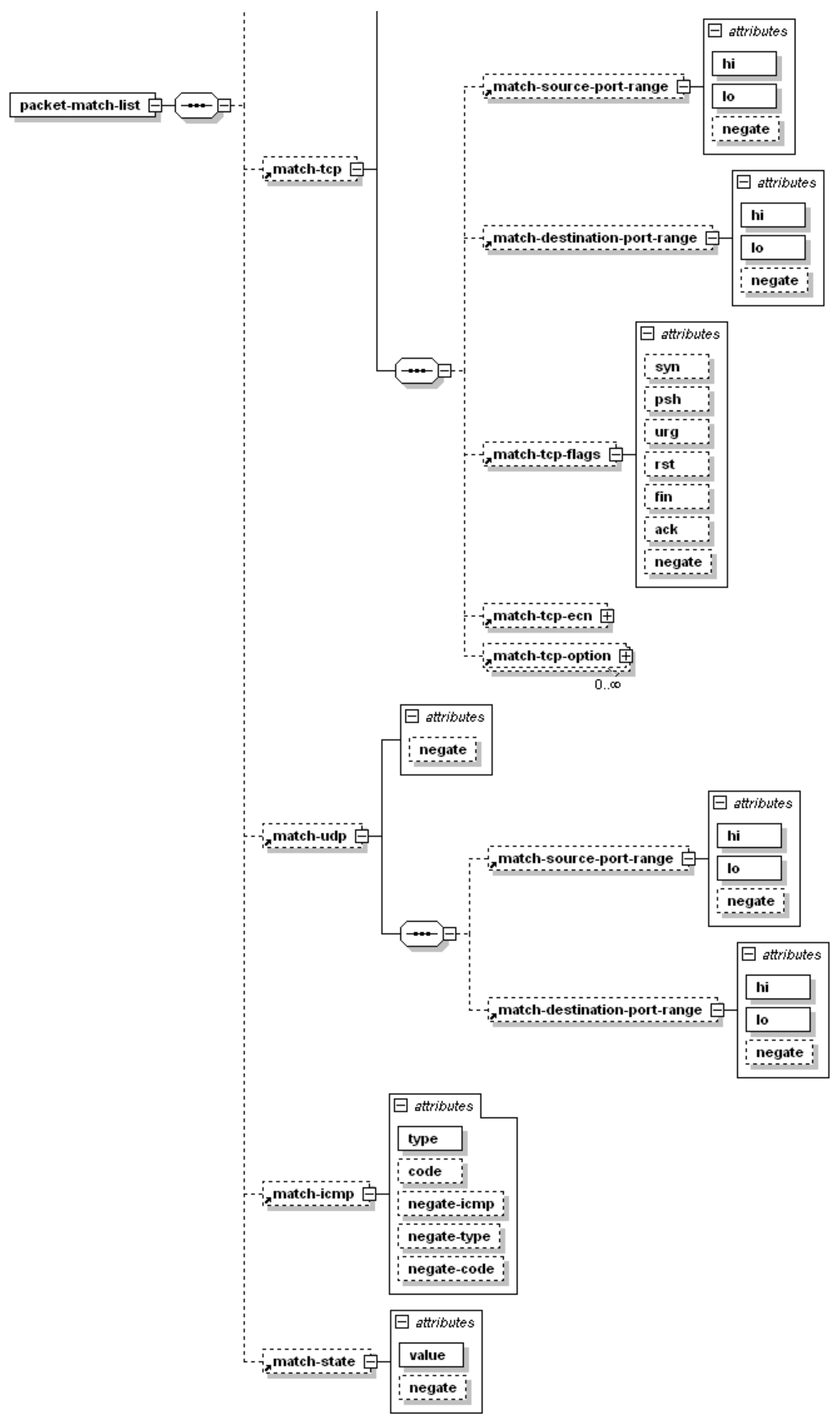


B.3.3 packet-filter-rule

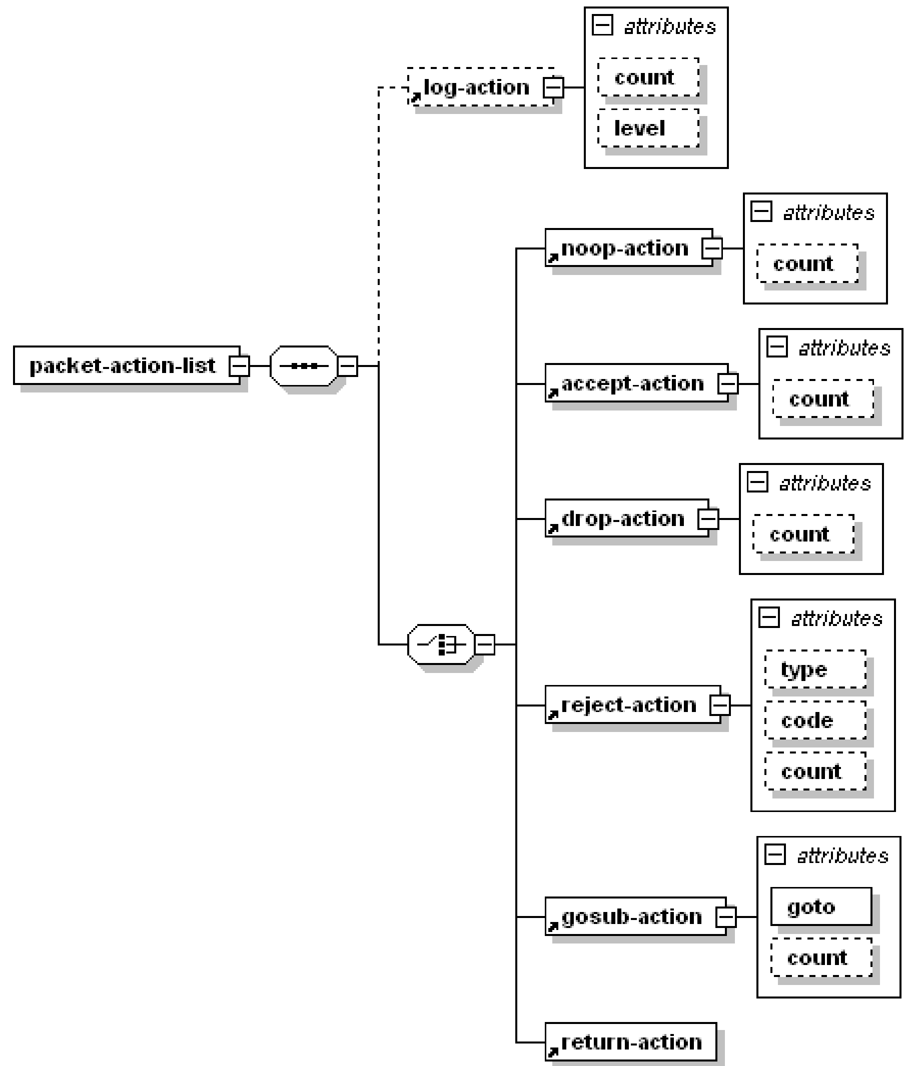


B.3.4 packet-match-list




packet-match-list
match-tcp
match-source-port-range
attributes
hi
lo
negate
match-destination-port-range
attributes
hi
lo
negate
match-tcp-flags
attributes
syn
psh
urg
rst
fin
ack
negate
match-tcp-ecn
match-tcp-option
0..∞
match-udp
attributes
negate
match-source-port-range
attributes
hi
lo
negate
match-destination-port-range
attributes
hi
lo
negate
match-icmp
attributes
type
code
negate-icmp
negate-type
negate-code
match-state
attributes
value
negate

B.3.5 packet-action-list



B.4 Extendended Events Schema from events.xsd

```
1  <xs:element name="packet-filter-rule">
2   <xs:complexType>
3    <xs:sequence>
4     <xs:element name="match-in-interface" minOccurs="0">
5      <xs:complexType>
6       <xs:attribute ref="negate"/>
7       <xs:attribute name="interface" type="xs:string"/>
8      </xs:complexType>
9     </xs:element>
10    <xs:element name="match-out-interface" minOccurs="0">
11     <xs:complexType>
```

```
12       <xs:attribute ref="negate"/>
13       <xs:attribute name="interface" type="xs:string"/>
14      </xs:complexType>
15     </xs:element>
16     <xs:element name="match-mac" minOccurs="0">
17      <xs:complexType>
18       <xs:attribute ref="negate"/>
19       <xs:attribute name="src" type="xs:string"/>
20      </xs:complexType>
21     </xs:element>
22     <xs:element ref="match-ipv4" minOccurs="0"/>
23     <xs:element ref="match-tcp" minOccurs="0"/>
24     <xs:element ref="match-udp" minOccurs="0"/>
25     <xs:element name="match-icmp" minOccurs="0">
26      <xs:complexType>
27       <xs:attribute ref="negate"/>
28       <xs:attribute name="type" type="xs:string" use="required"/>
29       <xs:attribute name="code" use="optional"/>
30       <xs:attribute name="negate-type" type="xs:string" use="optional"
            />
31       <xs:attribute name="negate-code" type="xs:string" use="optional"
            />
32      </xs:complexType>
33     </xs:element>
34     <xs:element name="match-state" minOccurs="0">
35      <xs:complexType>
36       <xs:attribute ref="negate"/>
37       <xs:attribute name="value" type="xs:string"/>
38      </xs:complexType>
39     </xs:element>
40    </xs:sequence>
41    <xs:attribute name="id" type="xs:string" use="optional"/>
42   </xs:complexType>
43  </xs:element>
44
45  <xs:element name="match-ipv4">
46   <xs:complexType>
47    <xs:sequence>
48     <xs:element name="match-source" type="iprange" minOccurs="0"/>
49     <xs:element name="match-destination" type="iprange" minOccurs="0"/
          >
50     <xs:element name="match-dsfield" minOccurs="0">
51      <xs:complexType>
52       <xs:attribute ref="negate" use="optional"/>
53       <xs:attribute name="dscp" type="xs:string" use="optional"/>
54       <xs:attribute name="dscp-class" type="xs:string" use="optional"/
            >
55      </xs:complexType>
56     </xs:element>
57     <xs:element name="match-ecnfield" minOccurs="0">
```

```
58       <xs:complexType>
59        <xs:attribute ref="negate"/>
60        <xs:attribute name="ect" type="xs:string" use="optional"/>
61        <xs:attribute name="ce" type="xs:string" use="optional"/>
62       </xs:complexType>
63      </xs:element>
64     </xs:sequence>
65     <xs:attribute name="fragment" type="xs:string"/>
66    </xs:complexType>
67   </xs:element>
68
69   <xs:element name="match-tcp">
70    <xs:complexType>
71     <xs:sequence>
72      <xs:element ref="match-source-port-range" minOccurs="0"/>
73      <xs:element ref="match-destination-port-range"  minOccurs="0"/>
74      <xs:element name="match-tcp-flags" minOccurs="0">
75       <xs:complexType>
76        <xs:attribute name="syn" type="xs:string" use="optional"/>
77        <xs:attribute name="psh" type="xs:string" use="optional"/>
78        <xs:attribute name="urg" type="xs:string" use="optional"/>
79        <xs:attribute name="rst" type="xs:string" use="optional"/>
80        <xs:attribute name="fin" type="xs:string" use="optional"/>
81        <xs:attribute name="ack" type="xs:string" use="optional"/>
82       </xs:complexType>
83      </xs:element>
84      <xs:element name="match-tcp-ecn" minOccurs="0">
85       <xs:complexType>
86        <xs:attribute name="ece" type="xs:string" use="optional"/>
87        <xs:attribute name="cwr" type="xs:string" use="optional"/>
88       </xs:complexType>
89      </xs:element>
90      <xs:element name="match-tcp-option" minOccurs="0" maxOccurs="
          unbounded">
91       <xs:complexType>
92        <xs:attribute ref="negate"/>
93        <xs:attribute name="kind" type="xs:string" use="required"/>
94       </xs:complexType>
95      </xs:element>
96     </xs:sequence>
97     <xs:attribute ref="negate"/>
98    </xs:complexType>
99   </xs:element>
100
101  <xs:element name="match-udp">
102   <xs:complexType>
103    <xs:sequence>
104     <xs:element ref="match-source-port-range" minOccurs="0"/>
105     <xs:element ref="match-destination-port-range" minOccurs="0"/>
106    </xs:sequence>
```

```
107     <xs:attribute ref="negate"/>
108   </xs:complexType>
109 </xs:element>
```

Appendix C

Examples

C.1 Properties file for Stateful Inspection

```
1  <firewall>
2    <statefulInspection>
3      <ip timeout="6000" />
4      <udp timeout="1800" />
5      <tcp>
6        <close_wait timeout="432000" />
7        <closed timeout="18000" />
8        <closing timeout="100" />
9        <established timeout="4320000" />
10       <fin_wait timeout="1200" />
11       <last_ack timeout="300" />
12       <listen timeout="1200" />
13       <none timeout="18000" />
14       <syn_rcv timeout="600" />
15       <close_sent timeout="1200" />
16       <time_wait timeout="1200" />
17     </tcp>
18   </statefulInspection>
19 </firewall>
```

C.2 Network Definition

C.2.1 Complete Network Definition

```
1  <nodes xmlns="http://diuf.unifr.ch/tns/projects/verinec/node"
2      xmlns:tr="http://diuf.unifr.ch/tns/projects/verinec/translation"
3      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4      xsi:schemaLocation="http://diuf.unifr.ch/tns/projects/verinec/
          node http://diuf.unifr.ch/tns/projects/verinec/node.xsd http:
          //diuf.unifr.ch/tns/projects/verinec/translation http://diuf.
          unifr.ch/tns/projects/verinec/translation.xsd">
```

```
5
6
7    <tr:type name='testtype' id='typ01'>
8       <tr:service name='ethernet' translation='linux-redhat' />
9       <tr:service name='serial' translation='wvdial' />
10      <tr:service name='dns' translation='bind8' />
11      <tr:service name='xy' translation='yz'>
12        <tr:target name="test">
13          <tr:wmi host="pc23" username="user@domain" />
14        </tr:target>
15      </tr:service>
16   </tr:type>
17
18   <variable name="session_wep" value="00
       :11:22:33:44:55:66:77:88:99:aa:bb" />
19   <variable name="session_wep_home" value="
       ff:ee:dd:cc:bb:aa:99:88:77:66:55:44" />
20
21 <node hostname="diufpc55">
22   <description>dominiks blder dell, auf dem X nicht luft</description
       >
23   <variable name="hostname" value="diufpc55" />
24
25   <prefix-lists>
26     <prefix-list name="trusted" id="pref1">
27       <description>IP numbers we trust</description>
28       <match-prefix address="192.168.0.0" length="16" />
29     </prefix-list>
30
31     <prefix-list name="trusted" id="pref2">
32       <description>A very evil hacker net</description>
33       <match-prefix address="10.10.30.0" length="24" />
34     </prefix-list>
35   </prefix-lists>
36
37   <hardware>
38     <ethernet name="Erste Ethernetkarte" hwaddress="00:10:ab:12:ff:23
         ">
39       <hint system="pc" slot="0" />
40       <hint system="junos" slot="0" pic="0" port="0" />
41       <ethernet-binding name="uni" id="xyz">
42         <nw id="i1" address="192.168.0.1" subnet="255.255.0.0"
             gateway="192.168.0.24"
43            type="ip">
44           <nw-dnsserver ip="192.168.0.254" />
45           <nw-dnsserver ip="192.168.0.253" />
46         </nw>
47         <nw id="i2" address="134.21.9.48" type="ip" />
48         <nw id="i3" hwaddress="00:10:ab:12:ff:12" address="
             134.21.9.49" type="ip" />
```

```
49        </ethernet-binding>
50      </ethernet>
51      <wlan name="Wirelesskarte" hwaddress="10:10:ab:12:ff:23">
52        <wlan-binding id="zyx" wepkey="$session_wep$" session="Wireless
              ">
53          <nw id="i10" address="90.160.84.12" type="ip" />
54        </wlan-binding>
55        <wlan-binding id="yzx" wepkey="$session_wep_home$" session="
              Home">
56          <nw id="i20" address="190.60.7.90" type="ip" />
57        </wlan-binding>
58      </wlan>
59      <serial name="altes 56k-Modem" devicenr="2">
60        <serial-binding id="pxx" phone="0041263008476" login="dominik"
              password="jungo" identifier="bluewin" protocol="ppp">
61          <nw id="i30" address="121.16.80.9" type="ip" />
62        </serial-binding>
63        <serial-binding id="pyy" phone="0041263008476" login="dominik"
              password="jungo" identifier="uni_RACE" protocol="slip">
64          <nw id="i40" type="ip" >
65            <dyn type="dhcp" timeout="10" retry="5" />
66          </nw>
67        </serial-binding>
68      </serial>
69    </hardware>
70
71    <services>
72
73      <routing/>
74
75      <dns>
76          <variable name="toonsdomain" value="toons.foo.net." />
77          <Zone match="$toonsdomain$"
78                type="master"
79                primaryns="ns.$toonsdomain$"
80                adminmail="root.$toonsdomain$"
81                serial="2004031700"
82                refresh="10800"
83                retry="3600"
84                expire="604800"
85                min_ttl="86400">
86
87            <dnsNS match="$toonsdomain$" target="$hostname$.$
                  toonsdomain$"/>
88
89            <dnsIPRange network="127.0.0">
90              <description>local</description>
91              <dnsA match="localhost" target="1"/>
92            </dnsIPRange>
93
```

```
94          <dnsIPRange network="192.168.15">
95            <description>network</description>
96            <dnsA match="$toonsdomain$" target="95"/>
97            <dnsA match="$hostname$" target="95"/>
98            <dnsA match="sly" target="90"/>
99            <dnsA match="ben" target="101"/>
100           <dnsA match="roadrunner" target="254"/>
101         </dnsIPRange>
102
103         <dnsIPRange network="192.168.30">
104           <description>network_1</description>
105           <dnsA match="$hostname$" target="95" />
106           <dnsA match="tigger" target="125" />
107           <dnsA match="kanga" target="150" />
108         </dnsIPRange>
109
110         <dnsCNAME match="www" target="ben" />
111         <dnsCNAME match="ftp" target="ben" />
112         <dnsCNAME match="www2" target="sly" />
113         <dnsCNAME match="mail" target="sly" />
114
115         <dnsMX match="$hostname$.$toonsdomain$" target="mail.$
              toonsdomain$" priority="10" />
116       </Zone>
117     </dns>
118
119     <dhcp>
120       <option-param param-name="domain-name" instance="toons.foo.net"
            />
121       <option-param param-name="domain-name-servers" instance="ns1.
            toons.foo.net, ns2.toons.foo.net"/>
122       <statement value="server-name diuf-dhcp"/>
123
124       <shared-network param-name="awesso-toons">
125         <statement value="filename boot"/>
126
127         <subnet addr="204.254.239.32" netmask="255.255.255.224">
128           <option-param param-name="domain-name" instance="toons.foo.
                net"/>
129           <option-param param-name="domain-name-servers" instance="ns
                .toons.foo.net"/>
130           <statement value="server-name dhcp-server"/>
131           <range begin="204.254.239.42" finish="204.254.239.62"/>
132         </subnet>
133       </shared-network>
134       <subnet addr="204.254.239.64" netmask="255.255.255.224">
135         <option-param param-name="domain-name" instance="toons.foo.
              net"/>
136         <option-param param-name="domain-name-servers" instance="ns.
              toons.foo.net"/>
```

```
137           <statement value="server-name dhcp-server"/>
138           <range begin="204.254.239.74" finish="204.254.239.94"/>
139         </subnet>
140         <group>
141           <statement value="routers 204.254.239.1"/>
142           <host host-name="host1">
143             <host-param card="ethernet" addr="00:c0:c3:cc:0a:8f"/>
144           </host>
145           <host host-name="host2">
146             <host-param card="ethernet" addr="00:c0:c3:2a:34:f5"/>
147           </host>
148         </group>
149       </dhcp>
150 
151       <packet-filters global-in="pfc2" global-out="pfc1">
152         <packet-filter-chain name="mychain" id="pfc1">
153           <deefault-policy>
154             <drop-action/>
155           </deefault-policy>
156 
157           <packet-filter-rule>
158             <packet-match-list>
159               <match-state value="NEW"/>
160             </packet-match-list>
161             <packet-action-list>
162               <accept-action/>
163             </packet-action-list>
164           </packet-filter-rule>
165 
166           <packet-filter-rule>
167             <packet-match-list>
168               <match-state value="ESTABLISHED"/>
169             </packet-match-list>
170             <packet-action-list>
171               <accept-action/>
172             </packet-action-list>
173           </packet-filter-rule>
174         </packet-filter-chain>
175 
176         <packet-filter-chain name="mychain" id="pfc2">
177           <deefault-policy>
178             <drop-action/>
179           </deefault-policy>
180 
181           <packet-filter-rule>
182             <packet-match-list>
183               <match-state value="ESTABLISHED"/>
184             </packet-match-list>
185             <packet-action-list>
186               <accept-action />
```

```
187             </packet-action-list>
188           </packet-filter-rule>
189         </packet-filter-chain>
190
191       </packet-filters>
192
193   </services>
194 </node>
195 </nodes>
```