| | |
|---|---|
| Project Number : | IST-2002-002154 |
| Project Title : | Distributed Adaptive Security by Programmable Firewall |

# DIADEM Firewall

# *D6 - Revised Interfaces Specification*

| | |
|---|---|
| Deliverable Type : | Document |
| Dissemination: | Public |
| Contractual date : | |

| | |
|---|---|
| Editor : | Dušan Gabrijelčič, Jozef Stefan Institute, (dusan@e5.ijs.si) |
| File Name: | Diadem Firewall – D6 – Revised Interfaces Specification.doc |
| Contributors : | See list of authors |
| Version : | Final |
| Version Date : | 24.1.2005 |
| Deliverable Status: | Deliverable D6 |

**The DIADEM Firewall consists of:**

| | Partner | Short name | Country |
|---|---|---|---|
| 1 | France Telecom | FT | France |
| 2 | University of Tübingen | TU | Germany |
| 3 | IBM Research GmbH Zurich Research Laboratory | IBM ZRL | Switzerland |
| 4 | Imperial College London | Imperial | United Kingdom |
| 5 | Jozef Stefan Institute | JSI | Slovenia |
| 6 | Groupe des Ecoles des Télécommunications | GET | France |
| 7 | Polish Telecom | TP | Poland |

**Project Management:**
Yannick Carlinet (FT)
Phone +33 2.96.05.03.25
Fax: +33 2 96 05 37 84
E-mail yannick.carlinet@francetelecom.com
France Telecom DAC/R2I
2 ave. Pierre Marzin,
22307 Lannion, France

**List of authors:**

Dušan Gabrijelčič, JSI
Yannick Carlinet, FT
Gerhard Muenz, TU
Falko Dressler, TU
Roland Wehage, TU
Sherif Yusuf, Imperial
Patricia Sagmeister, IBM ZRL
Gero Dittmann, IBM ZRL

**Executive summary**

This document describes the DIADEM firewall application programming interfaces (APIs) between the system elements defined by the DIADEM firewall architecture to support the operations and tasks defined in deliverable D5 [3]. The Monitoring API interfaces between Monitoring Element (ME) and Violation Detection (VD) to enable the VD to access monitored data and control its collection, exportation and aggregation in unified manner. The interface abstracts the hardware details of the specific Monitoring Device. The Notify API enables exchange of events in the system. The events are either notifications about the attacks or events that trigger the policies on the system elements. The System Manager can use the Service API to disseminate the policies to appropriate system elements. To counter the attacks, the System Manager can use events to trigger the policies on Firewall Elements (FE) or access their response mechanisms directly through Response API. The response mechanisms are based on response capabilities of Firewall Devices (FD) and the firewall API provides a device independent abstraction of the details of specific FDs. The response mechanisms are not always sufficient to mitigate new attacks. The Service API exported by the FE can be used to deploy code modules in the FDs to be able to mitigate them. On high-speed broadband connections, classification of the network traffic can be a major system bottleneck. The Classifier API enables the Firewall API to perform high speed classification in hardware and therefore achieve better performance. Each API is described with examples of usage that together support "Initial Demonstrator Specification" presented

in deliverable D7 [4].

**Acronyms**

| | |
|---|---|
| BEEP | Block Extensible Exchange Protocol |
| FD | Firewall Device |
| FE | Firewall Element |
| IDMEF | Intrusion Detection Message Exchange Format |
| IETF | Internet Engineering Task Force |
| IP | Internet Protocol |
| IPFIX | IP Flow Information eXport |
| MD | Monitoring Device |
| ME | Monitoring Element |
| PSAMP | Packet SAMPling |
| RPC | Remote Procedure Call |
| SCTP | Stream Control Transmission Protocol |
| SM | System Manager |
| SMI | Structure of Management Information |
| SNMP | Simple Network Management Protocol |
| SOAP | Simple Object Access Protocol |
| SSH | Secure Shell |
| TCP | Transmission Control Protocol |
| TS | Time Stamp |
| UDP | User Datagram Protocol |
| VD | Violation Detection |
| XML | eXtensible Markup Language |

**Table of Contents**

# 1 Introduction

The "Revised Interface specification" aims to revise a set of DIADEM APIs defined in "Initial Interface Specification", project deliverable D2 [2]. The APIs in the document are extended, further refined and complemented with examples of their usage. They were designed in a way to provide functionality required for operation of DIADEM distributed firewall architecture as defined in deliverable D5 [3]. The examples focus on project use cases as defined in "Initial Demonstrator Specification", deliverable D7 [7].
In the document the high level APIs are presented first: the Service API in section 2, the Response API in section 3 and the Notify API in section 4. As high level APIs they provide basic system services like event service, policy dissemination, extensibility of response mechanisms and accessibility of response actions. They rely on lower level APIs, namely the Monitoring, Firewall and Classifier API, which are presented in sections 5 to 7. They enable a usage of various devices for monitoring and firewall purposes and provide access to high speed classification in hardware.

# 2 Service API

The service API is a remote interface offered by an element, i.e. a Monitoring Element or a Firewall Element. This API allows the Violation Detection or the System manager to delegate part of their decision logic by transmitting policies to the elements. The API allows also to extend the functionalities (or capabilities) of the devices controlled by the element, and manage these extensions. The extension of the functionalities of a device is made thanks to modules. They are code archives that can be deployed and executed remotely by the element.

This API contains the following functionalities:
- Load/Remove policy
- Activate/Disable policy
- Load/Remove module
- Query capabilities of the element
- Query configuration/state of the element

## 2.1 Load/Remove policy

These operation are used either by the System Manager to deploy a policy on a Firewall Element and on Violation Detection Facilities or by the Violation Detection to deploy policies on a Monitoring Element. Once a policy is loaded, it must be activated before it is actually enforced. Once a policy is not used anymore, and is not planned to be used in the future, it can be removed from the element in order to free some resources (memory, disk space). The signatures of the functions are as follows:

- `loadPolicy(Policy p) returns LoadPolicyResult`

  where `LoadPolicyResult` is an object that contains the following fields:
  ```
  boolean success
  String errorMessage
  String identifier
  ```

The return type contains a Boolean that indicates if the policy could successfully be loaded. If its value is true the field `identifier` is set, and if its value is false, the field `errorMessage` contains an explanation of the problem.

- `removePolicy(String identifier) returns RemovePolicyResult`

  where `RemovePolicyResult` is an object that contains the following fields:
  ```
  boolean success
  String errorMessage
  ```

The return type contains a Boolean that indicates if the policy could successfully be loaded, and if its value is false, the field `errorMessage` contains an explanation of the problem.

### 2.2 Activate/Disable policy

Once activated, the policy is actually enforced on the element thanks to this function. When a policy should not be enforced anymore, it can be disabled through the provided function. The policy can later be totally removed of the element, or activated again. The signatures of the functions are as follows:

- `activatePolicy(String identifier) returns ActivatePolicyResult`
  where `ActivatePolicyResult` is an object that contains the following fields:
  `boolean success`
  `String errorMessage`

- `disablePolicy(String identifier) returns DisablePolicyResult`
  where `DisablePolicyResult` is an object that contains the following fields:
  `boolean success`
  `String errorMessage`

### 2.3 Load/Remove module

This function is used to load a module on an element. A monitoring module can be loaded on a Monitoring Element by the Violation Detection Facility, or a response module can be loaded on a Firewall Element by the System Manager. The component that uses this function must of course make sure beforehand that the element supports the loading of new modules (cf. 'query capabilities of the element' below). Once a module is loaded, the new functionality of the element can be used by a policy. When a module is not needed anymore, it can be removed from the element to free up resources. The functions are as follows:

- `loadModule(Module m) returns LoadModuleResult`
  in which Module is an object that contains a path to the code archive of the module. `LoadModuleResult` is an object with the following structure:
  - `boolean success` - set to true if the operation succeeds
  - `String errorMessage` - holds an error message, if applicable
  - `String id` - an identifier for the loaded module
- `removeModule(String id) returns RemoveModuleResult`
  `RemoveModuleResult` contains the field:
  - `boolean success` - set to true if the operation succeeds
  - `String errorMessage` - holds an error message, if applicable

### 2.4 Query capabilities of the element

Since a great diversity of Monitoring Devices or Firewall Devices can be part of the Diadem Firewall distributed architecture, it is sometimes necessary for the Violation Detection Facility or the System Manager to have information about the capabilities of these devices. Indeed it is useless to deploy a policy on an element if the underlying device does not have the ability to enforce it. On the other hand if a new module is loaded in the Firewall Device the capabilities of the device are extended.

- `listCapabilites()returns ListCapabilities`
  `ListCapabilities` is a list of Strings, each one describing a capability the element has
- `hasModuleEnvironment() returns boolean` – the return value is true if the element has a module environment available for the element.
- `addCapability(String capability)`

### 2.5 Query configuration/state of the element

This function returns information about the state of the element, such as: what policies are loaded, what policies are enforced, what modules are available, and general information about the element.

The element offers the following additional functions:

- `listLoadedPolicies() returns LoadedPolicyList`
- `listEnforcedPolicies() returns EnforcedPolicyList`
- `listLoadedModule() returns LoadedModuleList`

  where `LoadedPolicyList` and `EnforcedPolicyList` are lists of objects of type `Policy`, and `LoadedModuleList` is a list of objects of type `Module` (defined in subsection 2.3).
- `getConfig() returns ElementConfig`

  `ElementConfig` contains informatin such as the element name, network name, IP address, performance, what are the devices attached to this element, and so on.

## 2.6 Examples

Let us consider in this example a Firewall Element that has a router and an Open Firewall Device attached to it as shown in Figure 1.

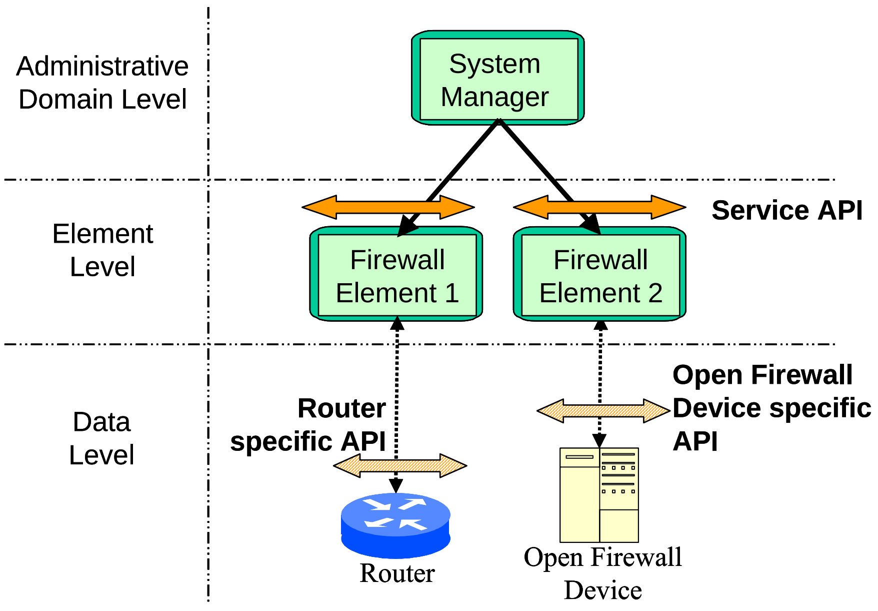

**Figure 1: example setup**

This example is in relation to the TCP SYN flood use-case described in details in D7 [4]. We suppose that an attack was detected and that the System Manager was notified of this attack. The System Manager now needs to install a reaction function on the Firewall Elements. The reaction function requires non-standard specialized processing on the traffic flows, so it cannot be implemented by the router. Therefore the System Manager must instruct the router to re-route the flows to the Open Firewall Device so that they can be processed. This is done as follows:

Let `TCPSynRedirect` be an object of type `Policy` that contains this semantics: if an IP packet is a TCP SYN message and has destination address $IP_{victim}$ then redirect it to $IP_{ODF}$ ($IP_{victim}$ and $IP_{ODF}$ represent respectively the IP address of the victim of the previously detected attack and the address of the Open Firewall Device).

The System Manager loads the policy by calling this function:
`loadPolicy(TCPSynRedirect)`
on the Firewall element. When the System Manager calls the function
`activatePolicy("TCPSynRedirect")`
the Firewall Element configures the router so that the considered traffic is actually redirected to the Open Firewall Device.

Since the processing to be performed on the redirected traffic is very specialized, and therefore is not available on the device, the system Manager now has to load a module that will contain the logic of this processing. We suppose this logic is stored by the System Manager as a Java archive of name 'm'. The System Manager calls the function
`loadModule(m)`
It also creates the policy `TCPSynProcessing`: if an IP packet is a TCP SYN message then apply the processing of module 'm' (please see [4] for a detailed description of this processing). Therefore calling
`loadPolicy(TCPSynProcessing)`
and
`activatePolicy("TCPSynProcessing")`
will have the Firewall Element configure the Open Firewall Device so that the latter actually perform the required processing.

This is summarized in the following sequence diagram:

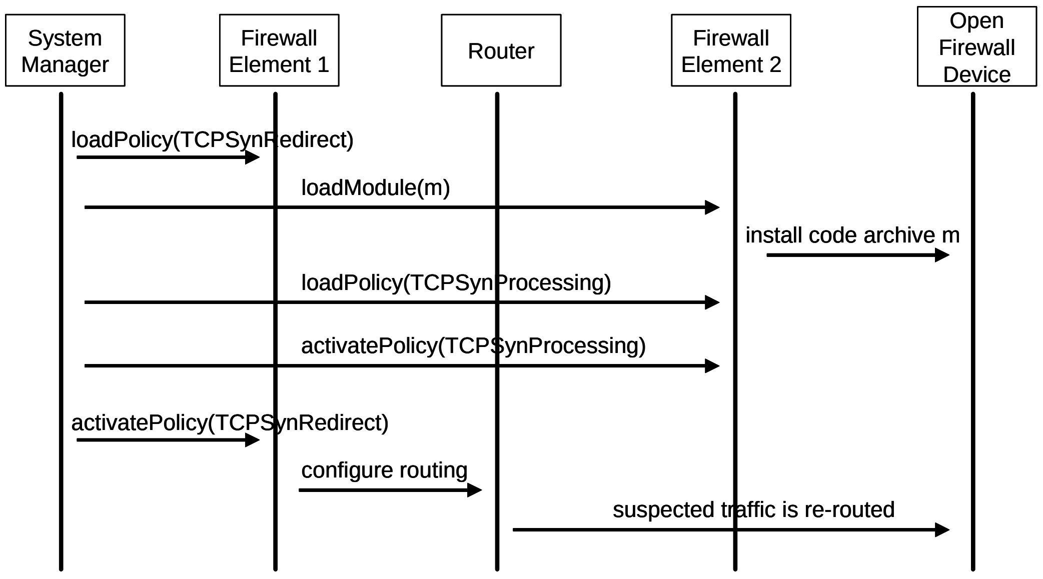
System Manager
Firewall Element 1
Router
Firewall Element 2
Open Firewall Device
loadPolicy(TCPSynRedirect)
loadModule(m)
install code archive m
loadPolicy(TCPSynProcessing)
activatePolicy(TCPSynProcessing)
activatePolicy(TCPSynRedirect)
configure routing
suspected traffic is re-routed

**Figure 2: Sequence diagram of an example showing the redirection of a flow, and its processing by the OFD**

# 3 Response API

This API defines specific actions that can be performed as the result of a notification being received. The response actions could be initiated by the System Manager, distributed local System Managers or even by an 'intelligent' firewall which detects and responds to attacks by itself. Defined below are typical APIs for some examples of such responses.

- Kill Session

<Source Address, Destination Address, Source Port, Destination Port, Sequence Number, Acknowledgement Number, Flag(s), Direction>

  - Flag(s): RESET flag must be set, at least for a TCP RESET packet
  - Direction: Either both ends of the connection will receive the RESET packet, or just the victim's machine. If both ends are to be sent RESET packets, then the source/destination addresses will be interchanged for the attacker's RESET packet

- Redirect - can be done with firewall capabilities or with manipulation of routes

<Protocol, Source Address, Destination Address, Source Port, Destination Port, Redirect Destination, Action>
<Destination Address, Redirection Type<Routing Algorithm >, Action>

  - Routing Algorithm: The redirection could be based on routing, where we specify the next hop of the packet, or we provide a static route via a routing table
  - Redirect Destination: Either for Blackhole (to be discarded) or Sinkhole ( to be logged) routing
  - Action: For Sinkhole, some data need to be logged

- Rate Limiting

<Protocol, Source Address, Destination Address, Source Port, Destination Port, Flag(s), Limit, Action>

  - Flag(s): Flag bits need to be set for some TCP protocol attacks, for example, SYN bit will be set in a TCP SYN flood attack
  - Limit: The threshold NOT to be exceeded. This limit could be number of packets, maximum bandwidth etc.
  - Action: We need to Drop packets after threshold is reached, but we may also want to Log them

Below are two examples of responses to a received notification of a TCP SYN attack. The response is simply to send a RST packet to both the source and target hosts. In the case where the source host does not handle RST packets in the normal way, we are still able to reset the connection with the RST packet sent to the target host, since this should handle the RST packet in the required manner.

When the response system receives the notification, it examines it to determine the appropriate action to undertake. Once the action is determined, the response system filters out the necessary information from the notification, and then constructs the response that is sent to the appropriate device that will carry out the action. The information required for the API can be found in deliverable D4.

### 3.1 Kill session

The first response that can be applied to this attack is to simply kill (disconnect) the attempted connection. The information required from the notification is the source of the attack, the target of the attack and some additional data, which in this case are IP packet header data. The rest of the elements in the API are specific to the response and the device that will carry out the action, for example, the <action> element states the action that needs to be performed by the device, in this case *kill_session* equates to *send RST* packet. In addition, the <direction> element states whom the RST packet should be sent to, the source, target or both hosts.

```
<response>
        <action> kill_session </action>
        <direction> both </direction>
        <source>
                <spoofed> no </spoofed>
                <interface> eth0 </interface>
                <node>
                        <category> unknown </category>
                        <name> achilles </name>
                        <address> 10.0.1.1 </address>
                        <location> DSE Lab </location>
                </node>
        </source>
        <target>
                <decoy> no </spoofed>
                <interface> eth1 </interface>
                <node>
                        <category> unknown </category>
                        <name> poseidon </name>
                        <address> 10.0.2.1 </address>
                        <location> DSE Lab </location>
                </node>
        </target>
        <additional_data>
                <transport_fields>
                        <tcp>
                                <src_seq> 40000 </src_seq>
                                <src_ack> 60000 </src_ack>
                                <dest_seq> 140000 </dest_seq>
                                <dest_ack> 160000 </dest_ack>
                        </tcp>
                </transport_fields>
                <network_fields>
                        <flags>
                                <syn> true </syn>
                        </flags>
                </network_fields>
        </additional_data>
</response>
```

### 3.2 Redirect

The second possible response that can be applied to a TCP SYN attack is to redirect the attempted connection away from the target, until we can confirm the validity of the source. The information required from the notification for redirection is the source of the attack, the target of the attack and some additional data, which in this case are IP packet header data. In addition, we require the type of redirection, which is different depending on the device that will carry out the redirection. The rest of the elements in the API are specific to the response and the device that will carry out the action, for example, the <action> element states the action that needs to be performed by the device, in this case *redirect* the packets. This <action> element is different to the one that is a child of the <redirection_type> element, which states whether the device should *log* the packet or not.

```
<response>
        <action> redirect </action>
        <redirection_type>
                <firewall> </firewall>
                <routing>
                        <routing_algorithm> </routing_algorithm>
                </routing>
                <action> </action>
        </redirection_type>
        <redirect_destination> </redirect_destination>
        <source>
                <spoofed> no </spoofed>
                <interface> eth0 </interface>
                <node>
                        <category> unknown </category>
                        <name> achilles </name>
                        <address> 10.0.1.1 </address>
                        <location> DSE Lab </location>
                </node>
        </source>
        <target>
                <decoy> no </spoofed>
                <interface> eth1 </interface>
                <node>
                        <category> unknown </category>
                        <name> poseidon </name>
                        <address> 10.0.2.1 </address>
                        <location> DSE Lab </location>
                </node>
        </target>
        <additional_data>
                <transport_fields>
                        <tcp>
                                <src_port> </src_port>
                                <dest_port> </dest_port>
                        </tcp>
                </transport_fields>
                <network_fields>
                        <protocol> tcp </protocol>
                </network_fields>
        </additional_data>
</response>
```

# 4 Notify API

## 4.1 IDMEF Message Format

Figure 3 describes the IDMEF message format that is used for the notification of attacks. It is a template that requires the data to be filled. Some of the objects will not have a value for some messages. Most of the object are self-explanatory, but below we give a brief description of the main component that need clarification.

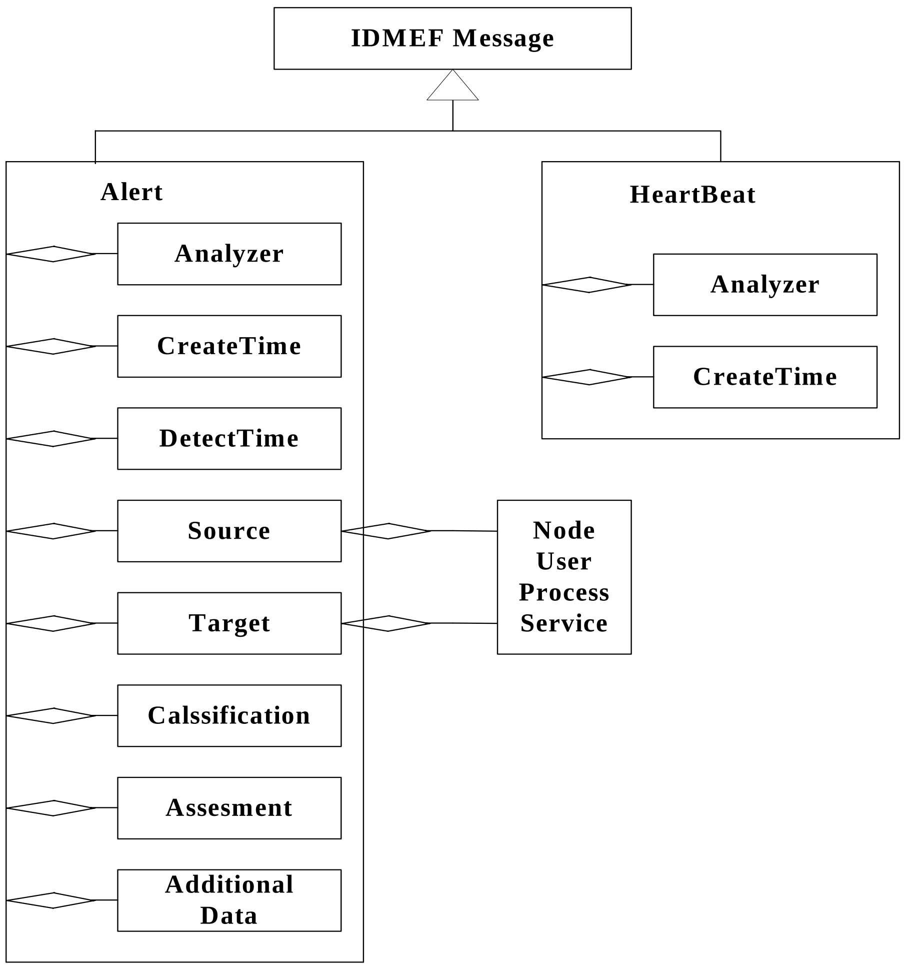

**Figure 3: IDMEF Message Format Structure**

The following is a description of some of the main elements:

**<idmef>**
The top most element of any XML description of the IDMEF message format

**<alert>**
One of the two types of messages supported. The alert message is for when there is a need to report that an incident has occurred or is occurring.

**<analyzer_id>**
A unique identifier for the analyzer (sensor) that generated the alert. This identifier only has to be unique in a single domain.

**<node>**
The node element provides additional information about the device, such as the network address, name, location etc.

**<create_time>**
The time the message is generated. This is not necessarily the same time as the time incident occurred.

**<detect_time>**
This is the time the incident is actually detected.

**<source>**
Details of the source(s) that generated the event, including name, address, location, user(s), process(es), interface it is seen on, if the address is spoofed etc.

**<target>**
Details of the possible target(s) of the generated event, including name, address, location, user(s), process(es), interface it is seen on, if the target is decoy etc.

**<classification>**
The classification provides the source from which the origin of the alert originates, and a url where more information about the alert can be found etc.

**<assessment>**
This element gives an analysis of the event that occurred. The impact it has (had) on the system, the confidence of the analyzer that the event is a genuine attack, and in some cases, where the analyzer can respond, the action taken by the analyzer.

**<additional_data>**
This is where users can make use of the scalability of the message format. Any extra information that can be obtained from the attack that is needed is included here.

**<heart_beat>**
This is the second type of message supported in IDMEF. It is used to notify the listening managers that the analyzer is still alive, but has nothing to report. This should be sent periodically at specified times. Basically, the absence of a heart beat message at the specified times can be seen as an event itself.

## 4.2 Notify API example

Here is an example use of the IDMEF format in a notification of a TCP SYN attack alert. Some of the elements from the full IDMEF XML have been omitted because they do not have any data and for clarity.

```
<notify>
        <idmef>
                <version> 1.0 </version>
                <alert>
                        <ident> dos_attack </ident>
                        <analyzer>
                                <analyzer_id> dse_lab_athena </analyzer_id>
                                <node>
                                        <category> unknown </category>
                                        <name> athena </name>
                                        <address>
                                                <category> ipv4-addr </category>
                                                <address> 10.0.1.2 </address>
                        </address>
                                        <location> DSE Lab </location>
                                </node>
                        </analyzer>
                        <source>
                                <spoofed> no </spoofed>
                                <interface> eth0 </interface>
                                <node>
                                        <category> unknown </category>
                                        <name> achilles </name>
                                        <address>
                                                <category> ipv4-addr </category>
                                                <address> 10.0.1.1</address>
                                        </address>
                                        <location> DSE Lab </location>
                                </node>
                                <user>
                                        <user_id> sy99 </user_id>
                                </user>
                                <process>
                                        <process_id> telnet <process_id>
                                </process>
                                <service>
                                        <name> netcat </name>
                                </service>
                        </source>
                        <target>
                                <decoy> no </spoofed>
                                <interface> eth1 </interface>
                                <node>
                                        <category> unknown </category>
                                        <name> poseidon </name>
                                        <address>
                                                <category> ipv4-addr </category>
                                                <address> 10.0.2.1 </address>
                                        </address>
                                        <location> DSE Lab </location>
                                </node>
                                <user>
                                        <user_id> sy99 </user_id>
                                </user>
                                <process>
                                        <process_id> day_time_service <process_id >
```

```
                    </process>
                    <service>
                              <name> netcat </name>
                    </service>
          </target>
          <classification>
                    <origin> unknown </origin>
                    <name> tcp_syn_attack </name>
          </classification>
          <assessment>
                    <impact>
                              <severity> medium </severity>
                              <completion> succeeded </completion>
                              <type> dos </type>
                    </impact>
                    <action>
                              <category> other </category>
                    </action>
                    <confidence>
                              <rating> high </rating>
                    </confidence>
          </assessment>
          <additional_data>
                    <transport_fields>
                              <tcp>
                                        <src_seq> 40000 </src_seq>
                                        <src_ack> 60000 </src_ack>
                                        <dest_seq> 140000 </dest_seq>
                                        <dest_ack> 160000 </dest_ack>
                              </tcp>
                    </transport_fields>
          </additional_data>
     </alert>
  </idmef>
</notify>
```

## 4.3 Elvin Content Based Messaging

We have decided to adopt the Elvin event notification service for the event notification in the Diadem project. Therefore the Notify API for will be those defined by the Elvin system, for example send, receive etc. Elvin is a content based messaging service that routes messages from one location to another, based entirely on the content(s) of each message. Elvin provides an extremely simple, flexible and secure communications infrastructure.

Elvin provides generic libraries that is linked by the client program and provides functions or classes used to access the Elvin4 server. To send notifications in Java for example, a producer has to build a *Notification* message, and emit the notification as follows:

<notify(*Notification n*)>

The *Notification* message is created with multiple Name/Value pairs. The name is always of type *String*, and the value file can be either a *String* or *any Literal*, for example:

<*String, Integer*>
<*String, Long*>
<*String, Double*>

Elvin uses client-server architecture for delivery of notifications. Clients establish sessions with an Elvin server process and are then able to send notifications for delivery or register to receive notifications sent by others. Clients can act as both producers and consumers of information within the same session.

The Elvin server manages client connections and routes notifications from producers to consumers. A consumer interested in a notification registers a subscription with the server; this subscription expresses selection criteria in terms of the content of each message. When the server receives a notification, it checks the content of the message against all registered subscriptions and forwards the notification to each client with a matching subscription. A single notification can match any number of subscriptions and is delivered to all active clients with a match.

A consumer receives notifications by adding a subscription to the Elvin server, with the following interface:

<addSubscription(Subscription sub)>

The *Subscription* from the consumer is written using the Elvin subscription language. An example of a subscription is:

(attack = = "tcp syn flood" && sensor_id = = "10.0.1.2" && regex(user, "[Ss]y99")

Basically, the subscription uses Name/Value pairs from the notification message, and/or it could be based on a regular expression. The Elvin subscription language provides a number of functions, which can be used in the subscription. Examples of such functions are:

The string comparison functions test strings in various ways. They test the first argument, which must name a value in the notification, to see if it matches any of the subsequent arguments.

*begins-with(name, string, ...)*: returns true if the value of name is a string and begins with any of the string arguments, *bottom* if the value name is not a string or not present in the notification and false otherwise.

*contains(name, string, ...)*: returns true if the value of name is a string and it contains any of the string arguments, *bottom* if the value name is not a string or not present in the notification and false otherwise.

*ends-with(name, string, ...)*: returns true if the value of name is a string and ends with any of the string arguments, *bottom* if the value name is not a string or not present in the notification and false otherwise.

The key advantage of Elvin is that messages are not addressed to a single recipient like a letter, or even a specific group like a notice board. The Elvin server delivers unaddressed notifications based on the information they contain rather than the direction they are sent. Producers do not have to specify a recipient or group of recipients. Recipients themselves, who in general are more aware of what information they require anyway, select only the messages of interest through their subscription.

Notifications in Elvin are delivered based on their content so the recipient must describe the messages of interest. Elvin requires consumers to provide a subscription, expressed in a simple syntax, which will match the events of interest. After establishing a connection, one or more expressions are sent to the server. While the consumer remains connected, all matching notifications are delivered via the connection.

Producers build a notification by inserting name-value pairs in a notification object. The name is specified as a string, and the value can be an integer, float, double or string. The subscription method is an expression that is also in the form of a name-value pair, written in Elvin's subscription language.

Currently, the subscription can only be registered based on simple value types such as integer, float, double, and string, with the corresponding name as a string. We are currently in discussion with the developers of Elvin, in an attempt to provide subscription based on querying XML documents using XPath XML query language.

## 5 Monitoring API

Monitoring Elements provide a common, abstract representation of the monitoring process by hiding device-specific implementation details of the underlying Monitoring Devices. The Monitoring API is used to configure and control the Monitoring Elements. A Monitoring Element translates Monitoring API calls into the corresponding device-specific commands of the associated Monitoring Devices. The Monitoring API also supports querying information about the status and device capabilities of the Monitoring Element. However, the transport of the monitoring data from the Monitoring Element to the Violation Detection system is not part of the Monitoring API, but is realized with the IPFIX protocol [5].

The Monitoring API is based on and realized with help of the Netconf protocol [6]. The following figure shows the relation between Violation Detection, Monitoring API and Monitoring Element. Netconf is employed for configuration and reconfiguration tasks while IPFIX is used for transporting the monitoring data to the Violation Detection.

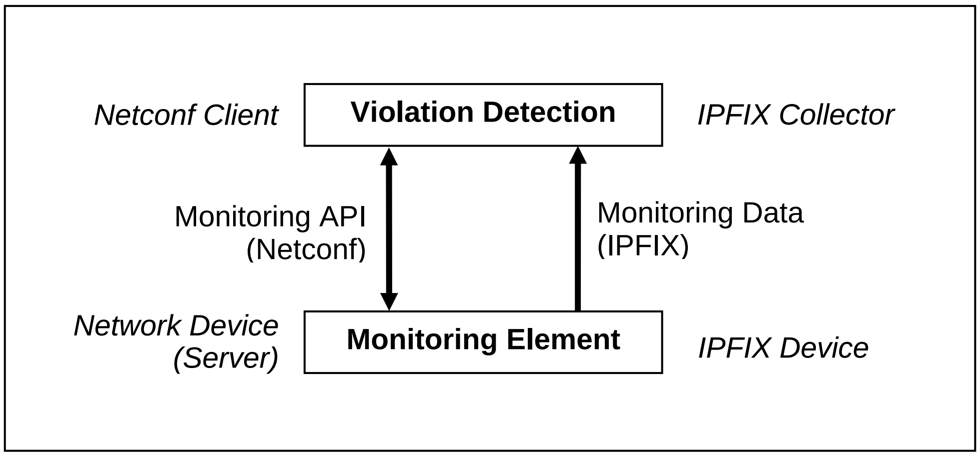

**Figure 4: Relations between Violation Detection, Monitoring Element and Monitoring API**

According to the abstract model of an IPFIX device as described in [7], the functionality of the Monitoring Element is divided into metering processes and exporting processes as shown below. In addition, an aggregation process is included as proposed in an Internet draft on IPFIX aggregation (work in progress).

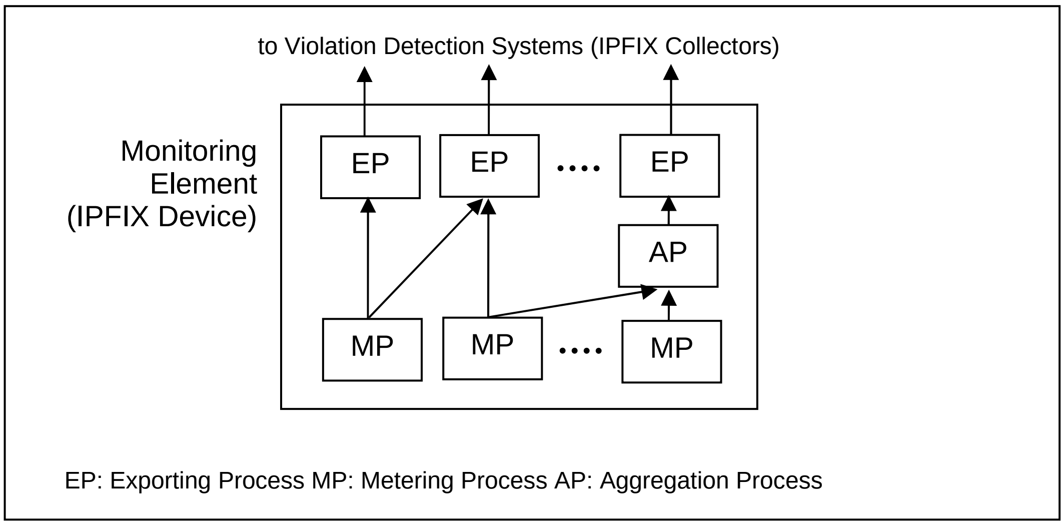
to Violation Detection Systems (IPFIX Collectors)
Monitoring Element (IPFIX Device)
EP
EP
EP
AP
MP
MP
MP
EP: Exporting Process MP: Metering Process AP: Aggregation Process

**Figure 5: Metering, Exporting and Aggregation Process**

A metering process gathers and optionally preprocesses monitoring data observed at an observation point (e.g. network interface). An exporting process transfers the monitoring data issued by one or more metering processes to the Violation Detection with help of the IPFIX protocol. Optionally, an aggregation process can be employed to compress IPFIX data in order to reduce the amount of monitoring data as well as the processing requirements at the Violation Detection.

Depending on the nature of the Monitoring Device, monitoring data consists of either IP flow records, records of metaflows, sampled packets or some other kind of information. Accordingly, the configurable parameters of the metering process also differ. In contrast, the configuration of the IPFIX exporting processes is identical for all kinds of monitoring data. It is possible to link metering processes of the different kinds to a single exporting process, i.e. an exporting process may export IP flow records and sampled packets at the same time. However, different templates have to be defined and used for the data export.

An aggregation process aggregates IP flow records provided by one or more associated metering processes and merges them into a single metaflow record. Like normal IP flow records, metaflow records are handed over to an exporting process for transfer to the Violation Detection. Though, the export of IP flow records and metaflow records usually requires different templates. Aggregation of sampled packets is not supported.

The remainder of this chapter is structured as follows. Sections 5.1 and 5.2 provide an introduction to the Netconf protocol and a short reference on Netconf operations. Section 5.3 specifies the XML structure of configuration data used for the Monitoring Elements. The usage of the Netconf based Monitoring API is finally illustrated with examples in section 5.4.

## 5.1 Netconf protocol

### 5.1.1 General Information

The IETF Network Configuration (Netconf) working group [8] is currently standardizing a protocol for the management and configuration of network devices (RFC expected at the beginning of 2005). Netconf protocol distinguishes between configuration data that can be uploaded, manipulated and retrieved from the device, and system state information that can be queried in read-only mode only. Netconf uses Extensible Markup Language (XML) for message encoding which allows using common known XML libraries and tools for message processing.

### 5.1.2 Remote procedure calls and operations

Netconf protocol is a client-server protocol which operations follow the style of remote procedure calls (RPCs). A message that the client sends to the network device represents an RPC invoking one out of a small set of basic operations. If the operation is successful, the network device responds sending a reply message containing a simple acknowledgement or some more data. In case of an error, the reply includes an error message. Netconf operations can be considered an API function set exposed by the network device.

While the set of operations is specified by the Netconf protocol, the actually transferred data is not. In terms of an API, this corresponds to function declarations that leave the parameter part open. This makes Netconf a very flexible protocol since the data can be defined according to the usage requirements. On the other hand, this leads to individual, vendor and device-specific solutions. There are some attempts in the IETF to define a Netconf data model that, like the information models and SMI definitions in SNMP-based network management, defines the structure and semantic of the exchanged data. However, it seems that there is no consensus yet, and no IETF draft has been published on this issue to date.

#### 5.1.3 Transport protocol requirements

The implementation of the Netconf protocol is not restricted to a specific transport protocol. However, the used transport protocol should meet the following requirements:

- The transport protocol has to provide for one or more multiple connection-oriented sessions.
- Since Netconf does not support any security mechanisms, the transport protocol has to provide necessary mechanisms for authentication, authorization, data integrity, privacy etc.

The Netconf working group has elaborated three alternative use cases that build Netconf either on SSH (secure shell), BEEP (Blocks Extensible Exchange Protocol) [9] or SOAP (Simple Object Access Protocol) [10].

#### 5.1.4 Configuration datastores

Netconf configuration operations are performed on so-called configuration datastores. A configuration datastore represents a complete set of configurable parameters of the managed device. A device has to provide at least the <running> configuration datastore that holds the currently active configuration. Additional datastores may be supported to define a specific <startup> configuration for the device or to store a <candidate> configuration that allows to edit a configuration step by step and commit it later to the <running> configuration as a whole.

#### 5.1.5 Netconf capabilities

Netconf specification is divided into the mandatory base functionality of the protocol and optional protocol extensions. When a new Netconf session is established, the two peers exchange <hello> messages that include information about their capabilities, i.e. the protocol extensions each of them supports.

Some proposed protocol extensions are:

- Writable running: The device supports direct writing to the <running> configuration datastore.
- Candidate configuration: The device supports the <candidate> configuration datastore.
- Rollback on error: The device allows automatically returning to the previous working configuration if the reconfiguration fails.
- Validate: The device can be instructed to check a configuration for syntactical and semantic errors before applying the configuration to the device.
- Distinct startup: A <startup> configuration datastore exists that defines the startup configuration of the device.
- URL: The device is able to upload and download the content of a configuration datastore to and from an external server.

### 5.2 Netconf Operations

An RPC sent from the client to the network device consists of one or more operations. Netconf specifies nine mandatory operations that have to be supported by the device, and some additional optional operations that may be supported by devices with special capabilities. The operations are briefly described in the following subsections. For further information refer to [6].

### 5.2.1 <get-config>

The <get-config> operation is used to retrieve all or parts of a specified configuration.
Parameters:

- source (mandatory): configuration datastore to be queried
- filter (optional): filter that identifies the portions of the configuration to be retrieved

Possible content of RPC reply:

- <data> element with the queried configuration data, or
- <rpc-error> element including some error information

### 5.2.2 <edit-config>

The <edit-config> operation changes a configuration datastore by creating, deleting or replacing all or parts of it. An optional `operation` attribute may be used to specify distinct actions for different elements of the datastore.
Parameters:

- target (mandatory): configuration datastore to be queried
- default-operation (optional): default action if `operation` attribute is missing
- test-option (optional): indicates if the configuration is to be validated before being written to the datastore (this option only applies to devices that support configuration validation)
- error-option (optional): specifies how to proceed on error
- config (mandatory): contains the configuration data

Possible content of RPC reply:

- RPC reply containing an <ok> element as an acknowledgement, or
- <rpc-error> element including some error information

### 5.2.3 <copy-config>

The <copy-config> operation copies the entire content of a configuration datastore to another configuration datastore. For example, a <candidate> configuration can be copied to the <running> configuration. If supported, configuration datastores can be also copied from and to external URLs.
Parameters:

- source (mandatory): source configuration datastore of the copy operation
- target (mandatory): target configuration datastore of the copy operation

Possible content of RPC reply:

- RPC reply containing an <ok> element as an acknowledgement, or
- <rpc-error> element including some error information

### 5.2.4 <delete-config>

The <delete-config> operation deletes a content of a configuration datastore. The <running> configuration cannot be deleted. If supported, the operation may also be applied on an external configuration datastore identified by an URL.
Parameters:

- target (mandatory): configuration datastore to be deleted

Possible content of RPC reply:

- RPC reply containing an <ok> element as an acknowledgement, or
- <rpc-error> element including some error information

### 5.2.5 <lock>

The <lock> operation locks a configuration datastore prohibiting write access by other clients.
Parameters:

- target (mandatory): configuration datastore to be locked

Possible content of RPC reply:

- RPC reply containing an <ok> element as an acknowledgement, or
- <rpc-error> element including some error information, e.g. that the lock is already held by another client

#### 5.2.6 <unlock>

The <unlock> operation releases a previously locked configuration datastore.
Parameters:

- target (mandatory): configuration datastore to be unlocked

Possible content of RPC reply:

- RPC reply containing an <ok> element as an acknowledgement, or
- <rpc-error> element including some error information

#### 5.2.7 <get>

The <get> operation is used to retrieve system configuration and state information.
Parameters:

- filter (optional): filter that identifies the portions of the configuration and state information to be retrieved

Possible content of RPC reply:

- RPC reply containing a <data> element with the queried information, or
- <rpc-error> element including some error information

#### 5.2.8 <close-session>

The <close-session> operation requests the termination of the Netconf session.
Parameters:

- none

Possible content of RPC reply:

- RPC reply containing an <ok> element as an acknowledgement, or
- <rpc-error> element including some error information

#### 5.2.9 <kill-session>

The <kill-session> operation kills another Netconf session. The nature of the session identifier parameter is not explained any further.
Parameters:

- session-id: identifier of the session to be killed

Possible content of RPC reply:

- RPC reply containing an <ok> element as an acknowledgement, or
- <rpc-error> element including some error information

## 5.3 Configuration of Monitoring Elements using Netconf

The Monitoring Elements export a Netconf interface for system configuration and management. The Monitoring API thus consists of Netconf operations and an XML representation for the monitoring-specific configuration and management data.

As mentioned in section 5, the functionality that has to be provided by Monitoring Elements and Devices can be divided into metering processes, aggregation processes, and exporting processes. The parameters that are needed to configure a metering process depend on the kind of monitoring data and the applied metering method. On the other hand, the configuration of an exporting process always comprises the same set of parameters, i.e. it does not dependent on the kind of monitoring data that is exported. Besides the configuration of exporting processes, the configuration of metering processes for IP flow metering, aggregation processes, and packet sampling is specified in the scope of this deliverable.
The Monitoring API uses a device-independent XML representation of configuration data. The following subsections provide a summary description of the used XML structure, while a formal specification using XML Schema description language can be found in appendix 1.

#### 5.3.1 Root element <monitorConfig>

The root element <monitorConfig> is an envelope for subelements dedicated to the configuration of metering processes, aggregation processes, sampling processes, and exporting processes. Further subelements are used to define lists of data and option templates that can be used by the exporting processes of the Monitoring Element.

### 5.3.2 Common types

This subsection briefly describes types that are not directly related to any kind of process.

- operation_type is a enumeration type used for the operation attribute. It is used in the context of <edit-config> operations. The usage of the operation attribute is explained in section 5.3.7.
- ipFilter_type contains the parameters of a IP packet filter. Types for other kinds of packet filters can be defined in an analogous manner. Filters are used in the context of metering and sampling process configuration.
- dataTemplate_type and optionTemplate_type allow to specify data templates and option templates used by the Monitoring Element.

### 5.3.3 Exporting process configuration types

The configurable parameters of an exporting process are described in [7]. The configuration parameters of the similar Netflow Version 9 protocol from Cisco can be found in the Netflow MIB [12]. Within the Monitoring API, the exporter_type is used for the configuration of an IPFIX exporting process. It covers the following settings:

- List of collectors the monitoring data is sent to.
- Identifier lists of the metering processes, sampling processes, and aggregation processes which monitoring data is exported.
- Maximum time to wait for more monitoring data to fill up an IPFIX packet.
- Parameters for periodical resending of templates to the collectors. (Note: If UDP is used as transport protocol, templates received by the collectors only have limited lifetime and have to be refreshed.)

### 5.3.4 IP flow metering process configuration types

The configuration of an IP flow metering process (meter_type) consists of the following settings [7][12]:

- Interface (observation point) of the monitoring device from where the monitoring data is captured.
- Time interval for periodical export of statistics of ongoing IP flows. This allows adapting the number of exported records per flow according to the needs of the recipients, i.e. the Violation Detection system.
- Time to wait until a flow expires when no more packets are observed. Upon expiration, a flow is removed from the list of ongoing flows.
- Packet filters for the selection of captured packets passed to the metering process.

### 5.3.5 Aggregation process configuration types

An aggregation process receives IP flow records from one or more metering processes and merges them into metaflows. The aggregation is controlled by aggregation rules. Some are implicitly defined while others can be configured.
Implicit rules concern:

- Timestamps (TS): TS of the first seen packet and TS of the last seen packet will be adopted to reflect the according meanings for the metaflow.
- Byte, packet, and connection counters: These counters are added to reflect the size of the metaflow and the number of connections merged into this metaflow.

Explicit rules concern the remaining attributes of a flow record. An explicit rule indicates the flow attributes used in the metaflow record and specifies a match pattern and an optional “keep” parameter for each of them. Flow records which attributes match all of the specified constraints are merged into one or more metaflows.

A match pattern can either be a single value, a range of values, or an “any” wildcard. A range of values can be specified for IP addresses and port lists:

- IP address range using a netmask: *address/mask*, e.g. 10.10.11.0/27

- TCP/UDP port list and range: *port1, port2, port3-port4*, e.g. 80,443,8080-8090

If the “keep” parameter is set for a specific flow attribute, the aggregation process results in multiple metaflows with different values for the corresponding attribute. Thus, flow records with different values for this attribute count for different metaflows. A single flow record may match several explicit rules. As a consequence, it counts for various metaflows.

The configuration of aggregation processes, provided by the aggregation_type, includes the following settings:

- List of metering processes which data records are to be aggregated.
- Set of explicit rules, each defining a list of flow attributes with associated match patterns and granularity parameters.

#### 5.3.6 Packet sampling configuration types

The IETF Packet Sampling (PSAMP) working group specifies an information model [11] and a MIB [13] that include the configurable parameters of sampling processes. Based on this, the Monitoring API provides the following settings in the sampler_type:

- Interface (observation point) of the monitoring device from where the monitoring data is captured.
- Applied sampling method and the settings of the corresponding parameters. The types selectAll_type, countBased_type, timeBased_type, randOutOfN_type, uniProb_type, nonUniProb_type, and flowState_type include the parameters of specific sampling algorithms. More information about sampling methods and parameters can be found in [11][13].

#### 5.3.7 Usage of the operation attribute

As mentioned in section 5.1.2, <edit-config> supports the usage of an optional operation attribute that indicates how the provided configuration data is to be inserted into the configuration datastore of the device. The operation attribute can be used with the following types: meter_type, sampler_type, aggregator_type, exporter_type, dataTemplate_type, optionTemplate_type, rule_type, collector_type, and ipFilter_type. Several instances of these elements may appear in the configuration datastore. In order to identify which instance an operation refers to, the usage of an id attribute is required.

If used with meter_type, sampler_type, aggregator_type, exporter_type, dataTemplate_type and optionTemplate_type, the settings of the operation attribute have the following meaning:

| Setting | Meaning | Configuration data |
|---|---|---|
| `Merge` | Default setting, apply the default operation indicated by <edit-config> and the individual settings of subelements if available. If no other operation is indicated replace any existing elements and create non-existing ones. | Configuration data needs not to be complete. |
| `replace` | Delete indicated instance and create new instance with the given configuration data. | Configuration data must be complete. |
| `create` | Create new instance with given configuration data. | Configuration data must be complete. |
| `delete` | Delete instance (identified by `id` attribute). | Configuration data is ignored. |

If used with collector_type, rule_type, and ipFilter_type, the operation attribute is ignored unless the operation attribute of the encompassing meter, sampler, aggregator or exporter element is set to `merge`. In this case, the meaning is as follows:

| Setting | Meaning | Configuration data |
|---|---|---|
| `Merge` | Same as `replace`. | Same as `replace`. |
| `replace` | Delete indicated instance und create new instance with given configuration data. | Configuration data must be complete. |
| `create` | Create new instance with given configuration data. | Configuration data must be complete. |
| `delete` | Delete instance (identified by `id` attribute). | Configuration data is ignored (can be omitted). |

## 5.4 Examples

The following examples illustrate how the Netconf messages in a typical configuration scenario look like. Note that "diadem-firewall.org/MonitoringAPI" is the name of the namespace all the elements and types mentioned above belong to.

### 5.4.1 Example: Configuration of an IP flow exporter

The following Netconf message is a request for the configuration of the current running configuration datastore, sent from the client to the network device. The configuration comprises a metering process, an aggregation process, an exporting process, and two templates:

- The metering process gathers statistics about TCP flows with destination 10.0.0.0/8.
- The aggregation process aggregates flow records with destination port 80 and destination 10.0.0.0/23, while maintaining different metaflow records per destination address. For example, two flows going to 10.0.0.10:80 but with different source addresses are merged into a metaflow with destination 10.0.0.10. A third flow going to 10.0.0.11:80 is counted in a separate metaflow with destination 10.0.0.11.
- The exporting process exports the flow records of the metering process and the metaflow records of the aggregation process to two distinct collectors, using UDP and SCTP as transport protocol for the IPFIX messages.
- The first template contains fields for source address (IPv4), source port, destination address (IPv4), destination port, and the packet counter. It is used for the export of flow records issued by the metering process. The second template contains destination address (IPv4), destination port, packet counter. It is used for the metaflow records of the results of the aggregation process.

```
<rpc message-id="101" xmlns="urn:ietf:params:xml:ns:netconf:base:1.0">
<edit-config>
 <source>
 <running />
 </source>
 <config>
       <monitorConfig xmlns="diadem-firewall.org/MonitoringAPI">
        <meters>
             <meter id="1" operation="create">
              <!-- metering process 1 maintains statistics about
              all TCP flows going to 10.0.0.0/8,
              flow records are exported using template 1 -->
              <templateId>1</templateId>
              <interface>eth0</interface>
              <exportPeriod>500</exportPeriod>
              <flowExpirationTime>10</flowExpirationTime>
              <filters>
                   <ipFilter id="1">
                    <dstAddress>10.0.0.0</dstAddress>
                    <dstAddressLength>8</dstAddressLength>
                    <protocol>6</protocol>
                   </ipFilter>
              </filters>
             </meter>
```

```
</meters>
<aggregators>
      <aggregator id="1" operation="create">
       <!-- aggregation process 1 accumulates flows measured
       by metering process 1 with dst port number 80, separated by
       destination address in the range 10.0.0.0/23,
       flow records are exported using template 2 -->
       <meterIds>1</meterIds>
       <rules>
             <rule id="1">
              <templateId>2</templateId>
              <match>
                    <type>11</type>
                    <pattern>80</pattern>
              </match>
              <match>
                    <type>12</type>
                    <pattern>10.0.0.0/23</pattern>
                    <keep />
              </match>
             </rule>
       </rules>
      </aggregator>
</aggregators>
<exporters>
      <exporter id="1" operation="create">
       <!-- exporting process 1 exports records from metering
       process 1 and aggregation process 1 to two distinct collectors -->
       <collectors>
             <collector id="1">
              <address>10.0.0.1</address>
              <port>1234</port>
              <protocol>udp</protocol>
             </collector>
             <collector id="2">
              <address>10.0.0.2</address>
              <port>1234</port>
              <protocol>sctp</protocol>
             </collector>
       </collectors>
       <meterIds>1</meterIds>
       <aggregatorIds>1</aggregatorIds>
       <dataSendTimeout>500</dataSendTimeout>
       <templateTimeout>10</templateTimeout>
       <templateRefreshRate>1000</templateRefreshRate>
      </exporter>
</exporters>
<dataTemplates>
      <dataTemplate id="1">
       <!-- template 1 contains source address (IPv4), source port,
       destination address (IPv4), destination port, packet counter -->
       <field>
             <type>8</type>
       </field>
       <field>
             <type>7</type>
       </field>
       <field>
             <type>12</type>
       </field>
       <field>
             <type>11</type>
       </field>
       <field>
             <type>2</type>
       </field>
      </dataTemplate>
      <dataTemplate id="2">
```

```
              <!-- template 2 contains destination address (IPv4), destination
              port, packet counter -->
              <field>
                   <type>12</type>
              </field>
              <field>
                   <type>11</type>
              </field>
              <field>
                   <type>2</type>
              </field>
             </dataTemplate>
        </dataTemplates>
       </monitorConfig>
 </config>
</edit-config>
</rpc>
```

The configuration request is acknowledged by the Monitoring Element with a positive response after successful configuration of the device:

```
<rpc-reply message-id="101" xmlns="urn:ietf:params:xml:ns:netconf:base:1.0">
 <ok />
</rpc-reply>
```

### 5.4.2 Example: Reconfiguration of a packet sampler

The following Netconf message requests the initialization of a new sampling process for time-based sampling of packets with destination address 10.0.0.5, the addition of a collector to the collector list of the exporting process, and a new data template for the export of sampled packets.

```
<rpc message-id="102" xmlns="urn:ietf:params:xml:ns:netconf:base:1.0">
<edit-config>
 <source>
 <running />
 </source>
 <config>
       <monitorConfig xmlns="diadem-firewall.org/MonitoringAPI">
        <samplers>
             <sampler id="1" operation="create">
              <!-- sampling process 1 applies time-based sampling on
              packets with destination address 10.0.0.5 -->
              <templateId>3</templateId>
              <interface>eth0</interface>
              <method>
                   <timeBased>
                    <interval>
                         123
                    </interval>
                    <spacing>
                         123
                    </spacing>
                   </timeBased>
              </method>
              <filters>
                   <ipFilter id="1">
                    <dstAddress>10.0.0.5</dstAddress>
                    <dstAddressLength>32</dstAddressLength>
                   </ipFilter>
              </filters>
             </sampler>
        </samplers>
        <exporters>
             <exporter id="1" operation="merge">
              <!-- sampling process 1 is associated to exporting process 1,
              and a new collector is added -->
              <collectors>
```

```
                     <collector id="3" operation="create">
                      <address>10.0.0.3</address>
                      <port>1234</port>
                      <protocol>tcp</protocol>
                     </collector>
               </collectors>
               <samplerIds>1</samplerIds>
              </exporter>
        </exporters>
        <dataTemplates>
              <dataTemplate id="3" operation="create">
               <!-- template 3 contains packet sequence number and sample -->
               <field>
                     <type>1025</type>
               </field>
               <field>
                     <type>1026</type>
               </field>
              </dataTemplate>
        </dataTemplates>
       </monitorConfig>
 </config>
</edit-config>
</rpc>
```

In this example, the Monitoring Element responds with an error message because it does not support time-based packet sampling (the format of error messages is described in [5]):

```
<rpc-reply message-id="102" xmlns="urn:ietf:params:xml:ns:netconf:base:1.0">
 <rpc-error>
 <error-type>application</error-type>
 <error-tag>OPERATION_NOT_SUPPORTED</error-tag>
 <error-severity>error</error-severity>
 <error-message>Sampling method timeBased not supported.</error-message>
 </rpc-error>
</rpc-reply>
```

# 6 Firewall API

Firewall API enables manipulation of firewall rules on multiple firewall implementations. Besides it can support routing tables and QoS manipulation as response mechanisms. It can be understood as a thin layer above the current/traditional firewall capabilities.

The basic ingredients of the API are a group, a rule and a selector. The **group** is a set of rules or/and groups. The group defines the manipulation of the network traffic that flows through a group. In some sense the group is similar to the UNIX file system structure where the groups correspond to directories and the rules to files. When the traffic flows through a group only the rules in the group are evaluated. The matched traffic by the selector can be redirected to a specific group if desired. Traditionally the groups are related to processing of the IP traffic and are grouped in groups like input, output and forward. In the sense of this API the groups can also be mapped to a Firewall Device or Firewall Element. Specified group operations, include *select*, *create, flush and list.* Abstract specification of the group is:

Group <group>:<group>

The <group> is a string. The only limitation to the name is that it has to be unique in the certain group. In contrast to the rules, as described below, groups enclosed in other groups, do not have any specific priority. The ruleid of the group is then reserved and equals a zero. The API supports three global variables: a ROOT_GROUP, a DEFAULT_GROUP and a CURRENT_GROUP. The ROOT_GROUP denotes a Firewall Element. The DEFAULT_GROUP is a default group for processing network traffic, such as a Firewall Device. The CURRENT_GROUP denotes a selected group by *select* function.

The **rule** defines the manipulation of a packet or a flow. The rule consists of a selector, selector action**,** log action and the time of the rule validity. Abstract specification of the rule is:

Rule <rule>:<selector> <selector_action> <log_action> [ "time" ]

Specified rule operations, we will call them the rule actions, are *insert*, *append* and *delete*. When the rules are created in the context of a group a sortable and unique rule identifier <ruleid> is assigned to the rule. Rule identifier is used to be able to insert or delete a rule at certain position in the group. The rule identifiers are positive integer numbers grater than zero.
The **selector** is a set of packet specifications that enables classification of a packet. Abstract syntax for the selector is:

Selector <selector>:<<selector 1> ... <selector n>>

The supported selectors are:

- source IP address/hostname: *source_ip*
- destination IP address/hostname: *dest_ip*
- the interface to match: *interface*
- the protocol to match: 'tcp' / 'udp' / 'icmp',... Can be a string or a number: *protocol*
- the source port of the packet: *source_port*
- destination port of the packet: *dest_port*
- TCP flags: tcp-flags:'fin', 'syn', 'rst', 'push', 'ack', 'urg', 'ecn', 'cwr': *tcp-flags*
- ICMP type: icmp-type: string or a code: *icmp-type*
- state: state: 'related', 'new', 'established', 'invalid': *state*

Source IP addresses can be host addresses or network address as combination of address and network mask. The interface denotes the packet incoming or outgoing interface. The source and destination ports can be numbers or expressions similar to those defined in classifier API in section 7.2. The protocol state matches the new state if the connection has been started, established for already known connection, related matches the packets that are not a part of a connection but related to it and the invalid matches the packets not related to any connection. The selector supports a wildcard matches that match for example any source or destination address, port, etc.

The operation on selected packet is a selector action. Currently specified selector actions are:

- accept the packet: *pass*
- drop the packet: *drop*
- return from the group: *return*
- reject the packet and optionally return icmp-error or TCP-reset: *reject [ <icmp-error> ]*
- Redirect the packet to the group or address: *redirect <queue> | <group> | <ip_address> [ protocol ] [ port ]*
- Rate limit the packet flow: *ratelimit <packets/s>|<kb/s>|<mb/s>*

The selector action pass accepts the packet and the drop action discards the packet. The redirect action redirects selected packets either to user space on Firewall Devices that support the action, group or to specific IP or network address. This action can be also used in conjunction with Module environment as described in [3] to deliver the packets to the code modules. The ratelimit action supports rate limiting of selected traffic to per packets, kilo or mega bytes per second.

Every rule can specify the log_action that enables logging of selected packets that pass the Firewall Device. The log action is specified as: *log [ log_level ] [log_mark][log_rate]*. The log level specifies the level of logging, like 'info', 'notice', 'warning', 'err', 'crit', 'alert' or 'emerg', log mark marks the log records and the log rate specifies the rate at which the logging records will be stored.

### 6.1 Operations on a group

**Group_select**

This function selects a group.

Syntax:

**group_select**(<group_name>)

Output:

The return values should follow the POSIX standard for appropriate file system operation (cd) in a meaningful way.

Input:

<group_name> is the name of the group being selected.

Notes:

The select must update the value of the CURRENT_GROUP.

**Group_create**

This function creates new group.

Syntax:

**group_create**(<group_name>)

Output:

The return values should follow the POSIX standard for appropriate file system operation (mkdir) in a meaningful way.

Input:

<group_name> is the name of the group being created. Group name can be also composed from more than one group followed by a new group.

Notes:

Certain firewalls have a set of pre defined groups, like iptables input/otput/forward. These groups should be created automatically and treated transparently by the API. If the group names can be composed, the default separator for the system should be defined. A special and global API variable can be used to define a separator. Additionally the function should support besides taking a group name as an argument to take a file as an argument. The file can contain a nested descriptions of all sub groups and their rules. The description itself should be as implementation independent as possible.

**Group_list**

This function returns the set of rules and groups in the group together with their ids.

Syntax:

**group_list**(<group_name>)

Output:

The function returns a list of rules <rule> and rule identifiers <ruleid>. Other return values should follow the POSIX standard for appropriate file system operation (ls) in a meaningful way.

Input:

<group_name> is the name of the group being listed.

Notes:

The recursive operations on the sub groups will be studied/added to the API if needed.

**Group_flush**

This function flushes all the rules in the group and deletes empty groups.

Syntax:

**group_flush**(<group_name>)

Output:

Other return values should follow the POSIX standard for appropriate file system operation (rm) in a meaningful way.

Input:

<group_name> is the name of the group being flushed

Notes:

### 6.2 Rule operations

**Rule_insert**

This function adds a rule to a CURRENT_GROUP.

Syntax:

**rule_insert**(<position>, <rule>)

Output:

The output of the function is a <ruleid>. Negative value is returned otherwise.

Input:

The <position> is the position of the rule after which the rule will be inserted. The <rule> is of a form as defined in section 0.

Example:

The following rule is inserted after the 10 rule in the current group.

**rule_insert**(10,protocol='tcp' state='new' redirect.group="TCP_SYNC")

Notes:

**Rule_delete**

This function deletes a rule in a CURRENT_GROUP.

Syntax:

**rule_delete**(<ruleid>)

Output:

The function returns the rule identifier of the rule before it, if successful, and if there is no rule before it, it returns 0. Negative value is returned otherwise.

Input:

The <ruleid> is the rule identifier of the rule to be deleted.

Example:

The rule with rule identifier 12 is deleted.

**rule_delete**(12)

Notes:

**Rule_insert** and **rule_delete** can be used to modify certain rule at certain position.

**Rule_append**

This function appends a rule in a CURRENT_GROUP.

Syntax:

**rule_append**(<rule>)

Output:

The function returns the rule identifier <ruleid> of the rule. Negative value is returned otherwise.

Input:

The <rule> is the rule to be appended.

Example:

**rule_append**(protocol='tcp' state='related' accept)

### 6.3 Communication in between Firewall Element and Firewall devices

The communication between a Firewall Element and the Firewall devices is part of the functionality implemented by the Firewall API. Multiple implementations are possible, because the API can support multiple Firewall Devices and such communication can be implemented in different ways.

We are evaluating Netlink protocol for communication of the Firewall Element with open firewall devices. Netlink is a Linux kernel protocol that provides datagram oriented service for communication in between user space processes and the Linux kernel. It consists of a standard socket based interface for user processes and an internal kernel API for kernel modules. The Netlink defines a family of protocols that enable a user to control various aspects of the kernel, like routing, IPv4 and IPv6 firewall, QoS, IP networking parameters and arp tables. As such is an excellent tool to enable the FE to access the kernel response capabilities and offer them through Firewall API as response mechanisms. Netlink, its protocols and some IP service templates are further described in [15]. Commercial firewalls and routers should be also supported either over CLI or, if available, on the top of functionality/protocol supported by selected device.

To enable the Firewall API to connect to a device the following function is used:

**attach_device**

This function starts the communication in between FE and FD. It enables specification of the protocol to be used for communication.

Syntax:

**attach_device**(ip_address, protocol)

Output:

The function returns a positive value on success and negative otherwise.

Input:

The input parameters are IP address of the device and the protocol to be used for communication.

Example:

Start communication with local kernel as Firewall Device.

**attach_device**(127.0.0.1, netlink)

Notes:

If the Firewall Element and the Firewall Device are closely coupled as in previous example the IP address of the device is its loopback address.

**detach_device**

This function ends the binding in between the Firewall Element and attached device. The function can be used to set the device in predefined or expected state.

Syntax:

**detach_device**(ip_address)

Output:

The function returns a positive value on success and negative otherwise.

Input:

The input parameter is IP address of the device.

Example:

Detach local kernel as Firewall Device.

**detach_device**(127.0.0.1)

## 7 Classifier API

This paragraph focuses on the further definition and refinement of the *abstract control API* for the Classifier Engine (CE), which has already been defined in D2 [2]. The described functions should be used for the communication between the Firewall Element and the Classifier Engine. The general structure of the abstraction layers is shown in Figure 6. Here, the upper layer software on a Firewall Element communicates with the Classifier Engine through the *Classifier API*. This API provides necessary functionality to configure the Classifier Engine and to control the packet-filter rule-set.

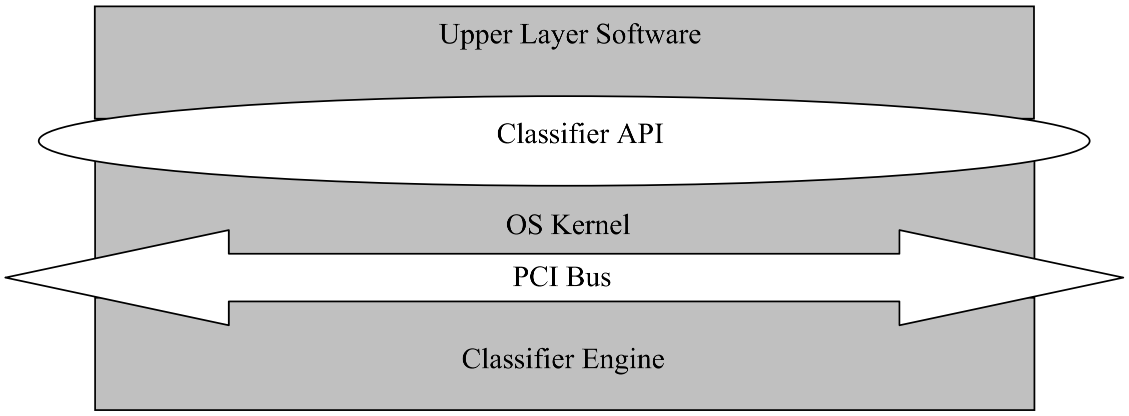

**Figure 6: Layer model**

The functions of the CE API are split in three categories: rule-specific functions, the search operation itself, and potentially a setup function. In the following paragraphs these functions are described in more detail.

### 7.1 Rule-specific functions

The category of rule-specific functions is responsible for dealing with the packet-filter rule-set. During the initialisation of the classifier engine these functions are used to transfer the current rule set to the CE. At runtime, these operations are necessary to perform dynamic updates of rules. There are four basic functions, which are supported:

- insert rule,
- delete rule,
- append rule, and
- modify rule.

Each of these functions can be mapped directly to a call of the firewall API. Before the rule specific functions can be explained in more detail we give an abstract description of a packet-filter rule.

## 7.2 Abstract packet-filter Rule description

Similar to the definition of a rule in the firewall API, a packet-filter rule R is a tuple of F components. While every component i, 0 < i < F, is a regular expression on the packet header field i, the last component of a packet-filter rule specifies the action or also called target. The action defines what to do with the packet that matched the current rule. The rule is expected in an according data structure.

Abstract syntax:

Rule R: <<component 1> ... <component F>><action>

The regular expressions allowed for a component are:

- <address> / <mask>
- <operator> <number>
- <number>
- <interface>
- <protocol>
- <tcp-flag>
- <icmp-type>
- <wildcard>

The address has to be within the range of the set of allowed IP addresses and can specify certain source or destination IP addresses. The mask can be given in decimal netmask notation or as the number of ones in the network mask. Possible examples for the address/mask combination would be:

192.168.0.0 / 255.255.255.0

192.168.0.0 / 24

The supported relations on the components are:

- equal: eq
- less then: lt
- greater then: gt
- range: rg <number> : <number>

Examples for those operator/number combinations would be:

- rg 20:21
- gt 1023

The *interface* is the input and/or output interface, such as eth0 or eth1. The protocol is the transport-protocol number in the IP header. Allowed TCP flags are fin, syn, rst, push, ack, urg, ecn, cwr. For ICMP it is possible to give the specific message type and if wanted the according code. Please note that states are not supported as packet-filter rule components.

In our examples, the syntax of the wildcard is: *

Further regular expressions for components depend on the number and the type of the fields that have to be matched to packet header fields. Possible further matches could be done on

- Start-of-connection (ACK) information for TCP packets
- MAC address

The last part of a rule is the action or target which defines the operation that has to be performed on the packet. Currently the following actions, listed in table, are supported:

| Action | Description |
|---|---|
| Drop | A packet that matches this rule is dropped immediately |
| Accept | A matching packet is accepted |
| Log Drop | The kernel logs some information on the matching packet and drops it immediately. |
| Log Accept | The kernel logs some information on the matching packet and accepts it. |
| Other Actions: TBD | TBD |
| User defined Actions | > 50 |

### 7.3 Insert rule

This function is used to dynamically insert a new packet-filter rule at runtime, or for setup purposes to load one or several initial filter rule sets. The ability to handle more than one packet-filter rule-set is an optional feature, which will not be implemented in our CE prototype. The insert function takes as input the new rule, a priority, and the number of the rule set in which this rule should be inserted. It returns the number of the rule in the rule set, the ruleID. If only one rule set is supported on the CE the <rule_set> parameter is obsolete.

Abstract syntax:

(<ruleID>, <error code>) = **insert_rule** (<priority>, <rule>, <rule_set>)

Output:

- <ruleID>:
  After inserting the rule in the current rule set, based on the priority given in the insert operation the rule gets a ruleID. This ruleID represents the position of the rule in the list of the updated rule set. It is used as a reference for this specific rule, for example for future modifications or updates of this rule.
- <error_code> :
  If the insertion was not successful an error code is returned, otherwise a success code is returned.

Input:

- <priority>:
  This gives the priority of the rule within the current rule set and according to this priority the ruleID is assigned.

- <rule>:
  The packet-filter rule is defined according to the description above.

- <rule_set>:
  Number of the rule set in which the rule has to be inserted. This parameter is obsolete in the default case where only one rule set is supported.

Example: (Only for the default cause, where one rule set is supported) insert_rule(10, 192.168.0.0/255.255.255.0 152.163.80.1/0.0.0.0 gt 1023 * Drop)

### 7.4 Delete rule

The function delete_rule takes as input the ruleID of the rule that has to be deleted and deletes the whole rule from the current rule set.

Abstract syntax:

<error_code> = **delete_rule** (<ruleID>, <rule_set>)

Output:

- <error_code> :
  If the deletion was not successful an error code is returned, otherwise a success code is returned.

Input:

- <ruleID>:
  The ruleID is the number of the rule which has to be deleted from the current rule set. This number was calculated at the time of the insertion of the rule and given back then as a result.

- <rule_set>:
  Number of the rule set in which the rule has to be deleted. This parameter is obsolete in the default case where only one rule set is supported.

Example: (Only for the default cause where one rule set is supported)

delete_rule(10)

### 7.5 Append rule

The function append_rule appends a new packet-filter rule at the end of the current rule set. The appended rule has therefore the lowest priority. This operation could be substituted by the function insert_rule with lowest priority.

Abstract syntax:

(<ruleID>, <error code>) = **append_rule** (<rule>, <rule_set>)

Output:

- <ruleID>:
  After appending the rule at the bottom of the current rule set, the rule gets a certain ruleID. This ruleID represents the position of the rule in the list of the now updated rule set. It is used as reference for this specific rule for example for future modifications or updates of this rule.
- <error_code> :
  If appending was not successful an error code is returned, otherwise a success code is returned.

Input:

- <rule>:
  The packet-filter rule is defined according to the description above.

- <rule_set>:
  Number of the rule set where the rule has to be appended. This parameter is obsolete in the default case where only one rule set is supported.

Example: (Only for the default cause, where one rule set is supported) append_rule (192.168.0.0/255.255.255.0 152.163.80.1/0.0.0.0 * * Accept)

## 7.6 Modify rule

The function modify_rule overwrites an existing rule in the current rule set. This operation could be substituted through the combination of the two functions delete_rule and insert_rule.

Abstract syntax:

<error_code> = **modify_rule** (<ruleID>, <rule>, <rule_set>)

Output:

- <error_code> :
  If the modification was not successful an error code is returned., otherwise a success code is returned.

Input:

- <ruleID>:
  The ruleID is the number of the rule which has to be modified in the current rule set. This number was calculated at the time of the insertion of the rule and given back then as a result.

- <rule>:
  The packet-filter rule is defined according to the description above.

- <rule_set>:
  Number of the rule set in which the rule has to be modified. This parameter is obsolete in the default case where only one rule set is supported.

Example: (Only for the default cause, where one rule set is supported)

modify_rule (11, 192.168.0.0/255.255.255.0 152.163.80.1/0.0.0.0 * * Accept)

## 7.7 Search function

The search function initiates the search for matching rules on the CE. Based on the input fields, such as source address, destination port number, etc., which have to be extracted from the packet header, the CE performs the packet classification. The returned result is the number of the matching rule, the so called <ruleID>, and the action or target of the matching rule. The CE expects already parsed header fields as inputs. Similar to the rules, the header fields are expected to be passed in a data structure.

Abstract syntax:

(<ruleID>, <error code>, <action>) = **search** (<header field 1> ...<header field N>, <rule_set>)

Output:

- <ruleID>:
  The ruleID is the number of the matching rule in the current rule set. This number was calculated at the time of the insertion of the rule and given back then as a result.
- <error_code> :
  If the search was not successful an error code is returned, otherwise a success code is returned.
- <action>: The currently supported actions are listed in the table above.

Input:

- <header field n>:
  The header field is a part of the packet header, which is used for the match against the rules in the current packet-filter rule set. The width depends on the type of the specific field.

- <rule_set>:
  Number of the rule set where the search has to be performed. This parameter is obsolete in the default case where only one rule set is supported.

Examples for commonly used header fields are:

- Source address, 32 bit
- Destination address, 32 bit
- Transport-protocol number, 8 bit
- Source port, 16 bit
- Destination port. 16 bit
- SYN bit

Example: (Only for the default cause, where one rule set is supported.)

search (192.163.190.69 152.163.80.1 1024 tcp)

Notes:

This function is used internally in the Firewall Element to initiate the classification of a packet.

## 7.8 Setup function

The setup function is used for the initialization of the CE and the according PCI driver.

Abstract syntax:

<error_code> = **setup** (<packet header start> <offset> <header field 1> <width 1> ...<header field N> <width N>)

Output:

- < error_code > :
  If the setup was not successful an error code is returned.

Input:

The whole input is expected to be in a certain data structure.

- <packet header start>:
  This is the position of the first header field in memory, i.e. the pointer to packet header start.
- <offset>:
  This is the according offset to packet header start.
- <header field>:
  Type of the header field which has to be used for packet classification.
- <width>:
  This is the width of the according header field.

# 8 Conclusions

In the document we have revised six DIADEM application programming interfaces (APIs): Service, Response, Notify, Monitor, Firewall and Classifier API. The APIs provide functionality in the network and on the system elements that enables operation of the DIADEM distributed firewall architecture as defined in D5 [3]. Monitoring API enables configuration of the Monitoring Elements and Monitoring Devices and collection, exportation and aggregation of monitored data. Notify API provides a way to deliver the attack notifications generated in Violation Detection to the System Manager and event service to trigger policies in the system. The System Manager can mitigate detected attacks with the Response API exported by Firewall Elements. The Response API uses Firewall API to access response mechanisms of the Firewall Devices in a unified way disregard of the Firewall Device type. On the end the Service API enables the System Manager to deploy, control and remove the policies or code modules to Firewall and Monitoring Elements. The Service API also provides the functions to query elements configuration and capabilities.
The presented APIs support the goals of the distributed firewall architecture as foreseen in DIADEM project: distributed monitoring and aggregation of network data, flexible and simple exchange of notifications and events in the system, centralized, hierarchical or distributed management of the elements with possibility of policy deployed and interpretation, extensibility of elements with new response mechanisms with deployment of new code modules, ability to integrate in the infrastructure existing devices for monitoring and firewall tasks and possibility of high speed classification in hardware to provide response mechanisms on broadband connections.
The APIs are illustrated with examples that together support the planned demonstration of the DIADEM firewall use cases as are defined in D7 [4].

## 9 References

[1] Distributed Adaptive Security by Programmable Firewall, DIADEM Firewall Technical Annex, August, 2003.

[2] Initial interface specification, DIADEM Firewall deliverable D2, July 2004.

[3] Architecture Specification, DIADEM Firewall deliverable D5, 2005.

[4] Initial Demonstrator Specification, DIADEM Firewall deliverable D7, 2005.

[5] B. Claise, "IPFIX Protocol Specification", draft-ietf-ipfix-protocol-05, August 2004.

[6] R. Enns, "NETCONF Configuration Protocol", draft-ietf-netconf-prot-04, October 2004.

[7] J. Quittek et al., "Requirements for IP Flow Information Export (IPFIX)", RFC 3917, October 2004.

[8] IETF Network Configuration (Netconf) working group, homepage: http://www.ietf.org/html.charters/netconf-charter.html.

[9] M. Rose, "The Blocks Extensible Exchange Protocol Core", RFC 3080, March 2001.

[10] Box, D., Ehnebuske, D., Kakivaya, G., Layman, A., Mendelsohn, N., Nielsen, H., Thatte, S. and D. Winer, "Simple Object Access Protocol (SOAP) 1.1", W3C Note NOTE-SOAP-20000508, May 2000, http://www.w3.org/TR/2000/NOTE-SOAP-20000508].

[11] T. Dietz et al., "Information Model for Packet Sampling Exports", draft-ietf-psamp-info-02, July 2004.

[12] Cisco Netflow MIB, Revision 200401090000Z, downloaded from http://www.cisco.com.

[13] T. Dietz et al. "Definitions of Managed Objects for Packet Sampling", draft-ietf-psamp-mib-03, July 2004.

[14] T. Zseby et al., "Sampling and Filtering Techniques for IP Packet Selection", draft-ietf-psamp-sample-tech-04, February 2004.

[15] J. Salim, H. Khosravi, A. Kleen, A. Kuznetsov, Linux Netlink as an IP Services Protocol, RFC 3549, July 2003, http://www.ietf.org/rfc/rfc3549.txt.

# 10 Appendix

## 1 XML Schema Definition of Monitoring API

### 1.1 Monitoring Element Configuration (monitor-config.xml-schema.xsd)

This XML Schema defines the XML root element of the configuration data structure <monitorConfig> that is used in Netconf messages. Further type definitions are grouped in four files that are imported at the beginning of this schema:

- common-types-xml-schema.xsd (see appendix 1.2)
- meter-types-xml-schema.xsd (see appendix 1.3)
- sampler-types-xml-schema.xsd (see appendix 1.4)
- exporter-types-xml-schema.xsd (see appendix 1.5)

The string “diadem-firewall.org/MonitoringAPI” serves as a unique identifier for the namespace of the XML Schema. It does not represent a valid URL.

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
      targetNamespace="diadem-firewall.org/MonitoringAPI"
      xmlns="diadem-firewall.org/MonitoringAPI"
      elementFormDefault="qualified">

 <xsd:annotation>
      <xsd:documentation xml:lang="en">
       XML Schema for IPFIX/PSAMP monitoring element/device configuration.
      </xsd:documentation>
 </xsd:annotation>

 <xsd:include schemaLocation="./common-types-xml-schema.xsd" />
 <xsd:include schemaLocation="./exporter-types-xml-schema.xsd" />
 <xsd:include schemaLocation="./meter-types-xml-schema.xsd" />
 <xsd:include schemaLocation="./sampler-types-xml-schema.xsd" />

 <xsd:element name="monitorConfig">
      <xsd:complexType>
       <xsd:sequence>
            <xsd:element name="meters" minOccurs="0">
             <xsd:complexType>
                  <xsd:sequence>
                   <xsd:element name="meter" type="meter_type"
maxOccurs="unbounded"/>
                  </xsd:sequence>
             </xsd:complexType>
            </xsd:element>
            <xsd:element name="samplers" minOccurs="0">
             <xsd:complexType>
                  <xsd:sequence>
                   <xsd:element name="sampler" type="sampler_type"
maxOccurs="unbounded"/>
                  </xsd:sequence>
             </xsd:complexType>
            </xsd:element>
            <xsd:element name="aggregators" minOccurs="0">
             <xsd:complexType>
                  <xsd:sequence>
                   <xsd:element name="aggregator" type="aggregator_type"
maxOccurs="unbounded"/>
                  </xsd:sequence>
             </xsd:complexType>
            </xsd:element>
            <xsd:element name="exporters" minOccurs="0">
```

```
              <xsd:complexType>
                   <xsd:sequence>
                    <xsd:element name="exporter" type="exporter_type"
maxOccurs="unbounded"/>
                   </xsd:sequence>
              </xsd:complexType>
             </xsd:element>
             <xsd:element name="dataTemplates" minOccurs="0">
              <xsd:complexType>
                   <xsd:sequence>
                    <xsd:element name="dataTemplate" type="dataTemplate_type"
maxOccurs="unbounded" />
                   </xsd:sequence>
              </xsd:complexType>
             </xsd:element>
             <xsd:element name="optionTemplates" minOccurs="0">
              <xsd:complexType>
                   <xsd:sequence>
                    <xsd:element name="optionTemplate" type="optionTemplate_type"
maxOccurs="unbounded" />
                   </xsd:sequence>
              </xsd:complexType>
             </xsd:element>
        </xsd:sequence>
       </xsd:complexType>
 </xsd:element>

</xsd:schema>
```

## 1.2 Common Types Definition (common-types-xml-schema.xsd)

Defines :

- operation_type (used for operation attribute, see 5.3.7)
- dataTemplate_type
- optionTemplate_type
- templateField_type
- ipFilter_type

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
       targetNamespace="diadem-firewall.org/MonitoringAPI"
       xmlns="diadem-firewall.org/MonitoringAPI"
       elementFormDefault="qualified">

 <xsd:annotation>
       <xsd:documentation xml:lang="en">
        XML Schema of common types used for IPFIX/PSAMP configuration.
       </xsd:documentation>
 </xsd:annotation>

 <xsd:simpleType name="operation_type">
       <xsd:restriction base="xsd:string">
        <xsd:enumeration value="merge" />
        <xsd:enumeration value="replace" />
        <xsd:enumeration value="create" />
        <xsd:enumeration value="delete" />
       </xsd:restriction>
 </xsd:simpleType>

 <xsd:complexType name="dataTemplate_type">
       <xsd:sequence minOccurs="0">
        <xsd:element name="field" type="templateField_type" minOccurs="0"
maxOccurs="unbounded" />
       </xsd:sequence>
       <xsd:attribute name="id" type="xsd:unsignedInt" use="required" />
       <xsd:attribute name="operation" type="operation_type" use="optional" />
 </xsd:complexType>
 <xsd:complexType name="optionTemplate_type">
       <xsd:sequence minOccurs="0">
        <xsd:element name="scopeField" type="templateField_type" minOccurs="0"
maxOccurs="unbounded" />
        <xsd:element name="optionField" type="templateField_type" minOccurs="0"
maxOccurs="unbounded" />
       </xsd:sequence>
       <xsd:attribute name="id" type="xsd:unsignedInt" use="required" />
       <xsd:attribute name="operation" type="operation_type" use="optional" />
 </xsd:complexType>

 <xsd:complexType name="templateField_type">
       <xsd:sequence>
        <xsd:element name="type" type="xsd:unsignedInt" />
        <xsd:element name="enterprise" type="xsd:unsignedInt" minOccurs="0" />
       </xsd:sequence>
 </xsd:complexType>

 <xsd:complexType name="ipFilter_type">
       <xsd:sequence minOccurs="0">
        <xsd:element name="srcAddress" type="xsd:string" minOccurs="0" />
        <xsd:element name="srcAddressLength" type="xsd:unsignedInt" minOccurs="0" />
        <xsd:element name="dstAddress" type="xsd:string" minOccurs="0" />
        <xsd:element name="dstAddressLength" type="xsd:unsignedInt" minOccurs="0" />
        <xsd:element name="srcPortMin" type="xsd:unsignedInt" minOccurs="0" />
        <xsd:element name="srcPortMax" type="xsd:unsignedInt" minOccurs="0" />
        <xsd:element name="dstPortMin" type="xsd:unsignedInt" minOccurs="0" />
        <xsd:element name="dstPortMax" type="xsd:unsignedInt" minOccurs="0" />
        <xsd:element name="protocol" type="xsd:unsignedInt" minOccurs="0" />
       </xsd:sequence>
       <xsd:attribute name="id" type="xsd:unsignedInt" use="required" />
       <xsd:attribute name="operation" type="operation_type" use="optional" />
 </xsd:complexType>

</xsd:schema>
```

| Element | Description | Attributes and subelements |
|---|---|---|
| `field, scopeField, optionField` | Field definition in a template (see [5] for more information). | `type:` numeric value of the field type<br>`enterprise:` IANA enterprise number for vender-specific types |
| `srcAddress, srcAddressLength, dstAddress, dstAddressLength, srcPortMin, srcPortMax, dstPortMin, dstPortMax, protocol` | IP filter 5-tuple. Omitted elements are considered as a wildcard. | `none` |

## 1.3 Metering Process Types Definition (meter-types-xml-schema.xsd)

Defines :

- meter_type
- aggregator_type
- rule_type

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
      targetNamespace="diadem-firewall.org/MonitoringAPI"
      xmlns="diadem-firewall.org/MonitoringAPI"
      elementFormDefault="qualified">
```

```
 <xsd:annotation>
      <xsd:documentation xml:lang="en">
       XML Schema of types for flow metering and aggregation process configuration.
      </xsd:documentation>
 </xsd:annotation>

 <xsd:complexType name="meter_type">
      <xsd:sequence minOccurs="0">
       <xsd:element name="templateId" type="xsd:unsignedInt" />
       <xsd:element name="interface" type="xsd:string" minOccurs="0" />
       <xsd:element name="exportPeriod" type="xsd:unsignedInt" minOccurs="0" />
       <xsd:element name="flowExpirationTime" type="xsd:unsignedInt" minOccurs="0"
/>
       <xsd:element name="filters" minOccurs="0">
            <xsd:complexType>
             <xsd:choice maxOccurs="unbounded">
                  <xsd:element name="ipFilter" type="ipFilter_type" />
             </xsd:choice>
            </xsd:complexType>
       </xsd:element>
      </xsd:sequence>
      <xsd:attribute name="id" type="xsd:unsignedInt" use="required" />
      <xsd:attribute name="operation" type="operation_type" use="optional" />
 </xsd:complexType>

 <xsd:complexType name="aggregator_type">
      <xsd:sequence minOccurs="0">
       <xsd:element name="meterIds" minOccurs="0">
            <xsd:simpleType>
             <xsd:list itemType="xsd:unsignedInt" />
            </xsd:simpleType>
       </xsd:element>
       <xsd:element name="rules" minOccurs="0">
            <xsd:complexType>
             <xsd:sequence>
                  <xsd:element name="rule" type="rule_type"
maxOccurs="unbounded"/>
             </xsd:sequence>
            </xsd:complexType>
       </xsd:element>
      </xsd:sequence>
      <xsd:attribute name="id" type="xsd:unsignedInt" use="required" />
      <xsd:attribute name="operation" type="operation_type" use="optional" />
 </xsd:complexType>

 <xsd:complexType name="rule_type">
      <xsd:sequence>
       <xsd:element name="templateId" type="xsd:unsignedInt" />
       <xsd:element name="match" minOccurs="0" maxOccurs="unbounded">
            <xsd:complexType>
             <xsd:sequence>
                  <xsd:element name="type" type="xsd:unsignedInt" />
                  <xsd:element name="enterprise" type="xsd:unsignedInt"
minOccurs="0" />
                  <xsd:element name="pattern" type="xsd:string" />
                  <xsd:element name="keep" minOccurs="0">
                   <xsd:complexType>
                   </xsd:complexType>
                  </xsd:element>
             </xsd:sequence>
            </xsd:complexType>
       </xsd:element>
      </xsd:sequence>
      <xsd:attribute name="id" type="xsd:unsignedInt" use="required" />
      <xsd:attribute name="operation" type="operation_type" use="optional" />
 </xsd:complexType>

</xsd:schema>
```

Description of the subelements in meter_type:

| Element | Description | Attributes and subelements |
|---|---|---|
| `templateId` | Identifier of the template to be used for exporting the flow records. | None |
| `interface` | Interface where packets are captured (observation point). | None |
| `exportPeriod` | Time interval (in microseconds) for periodical delivery of the statistics of ongoing flows to the exporting processes. | None |
| `flowExpirationTime` | If no more packets are observed during this time interval (in seconds), a flow is considered as expired. | None |
| `filters` | Packet filters that restrict the considered flows. | `ipFilter` or alternative filter |

Description of the subelements in aggregator_type:

| Element | Description | Attributes and subelements |
|---|---|---|
| `meterIds` | Identifer list of associated meters. | None |
| `rule` | Explicit aggregation rule. | `id:` rule identifier<br>`operation:` used in <edit-config><br>`templateId:` identifier of the template to be used for exporting the metaflow records<br>`match:` see below |
| `match` | Specifies a flow attribute, a match pattern and an optional "keep" parameter. | `type:` numeric value of the IPFIX field type<br>`enterprise:` IANA enterprise number for vender-specific types<br>`pattern:` match pattern<br>`keep:` "keep" parameter (empty element) |

### 1.4 Sampling Process Types Definition (sampler-types-xml-schema.xsd)

Defines :

- sampler_type
- selectAll_type
- countBased_type
- timeBased_type
- randOufOfN_type
- uniProb_type
- nonUniProb_type
- flowState_type

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
```

```
      targetNamespace="diadem-firewall.org/MonitoringAPI"
      xmlns="diadem-firewall.org/MonitoringAPI"
      elementFormDefault="qualified">

 <xsd:annotation>
      <xsd:documentation xml:lang="en">
       XML Schema of types for sampling process configuration.
       reference: draft-ietf-psamp-mib-03.txt
      </xsd:documentation>
 </xsd:annotation>

 <xsd:complexType name="sampler_type">
      <xsd:sequence minOccurs="0">
       <xsd:element name="templateId" type="xsd:unsignedInt" />
       <xsd:element name="interface" type="xsd:string" minOccurs="0" />
       <xsd:element name="method" minOccurs="0">
            <xsd:complexType>
             <xsd:choice>
                  <xsd:element name="selectAll" type="selectAll_type" />
                  <xsd:element name="countBased" type="countBased_type" />
                  <xsd:element name="timeBased" type="timeBased_type" />
                  <xsd:element name="randOutOfN" type="randOutOfN_type" />
                  <xsd:element name="uniProb" type="uniProb_type" />
                  <xsd:element name="nonUniProb" type="nonUniProb_type" />
                  <xsd:element name="flowState" type="flowState_type" />
             </xsd:choice>
            </xsd:complexType>
       </xsd:element>
       <xsd:element name="filters" minOccurs="0">
            <xsd:complexType>
             <xsd:sequence>
                  <xsd:element name="ipFilter" type="ipFilter_type"
maxOccurs="unbounded" />
             </xsd:sequence>
            </xsd:complexType>
       </xsd:element>
      </xsd:sequence>
      <xsd:attribute name="id" type="xsd:unsignedInt" use="required" />
      <xsd:attribute name="operation" type="operation_type" use="optional" />
 </xsd:complexType>

 <xsd:complexType name="selectAll_type" />

 <xsd:complexType name="countBased_type">
      <xsd:sequence>
       <xsd:element name="interval" type="xsd:unsignedInt" />
       <xsd:element name="spacing" type="xsd:unsignedInt" />
      </xsd:sequence>
 </xsd:complexType>

 <xsd:complexType name="timeBased_type">
      <xsd:sequence>
       <xsd:element name="interval" type="xsd:unsignedInt" />
       <xsd:element name="spacing" type="xsd:unsignedInt" />
      </xsd:sequence>
 </xsd:complexType>

 <xsd:complexType name="randOutOfN_type">
      <xsd:sequence>
       <xsd:element name="population" type="xsd:unsignedInt" />
       <xsd:element name="size" type="xsd:unsignedInt" />
      </xsd:sequence>
 </xsd:complexType>

 <xsd:complexType name="uniProb_type">
      <xsd:sequence>
       <xsd:element name="probability" type="xsd:unsignedInt">
            <xsd:annotation>
```

```
              <xsd:documentation xml:lang="en">
                   The given value must be divided by 4294967295
              </xsd:documentation>
             </xsd:annotation>
       </xsd:element>
      </xsd:sequence>
 </xsd:complexType>

 <xsd:complexType name="nonUniProb_type" mixed="true">
      <xsd:annotation>
       <xsd:documentation xml:lang="en">
             Non-uniform sampling parameters not specified here.
       </xsd:documentation>
      </xsd:annotation>
      <xsd:sequence>
       <xsd:any minOccurs="0" maxOccurs="unbounded" processContents="skip" />
      </xsd:sequence>
 </xsd:complexType>

 <xsd:complexType name="flowState_type" mixed="true">
      <xsd:annotation>
       <xsd:documentation xml:lang="en">
             Flow-state sampling parameters not specified here.
       </xsd:documentation>
      </xsd:annotation>
      <xsd:sequence>
       <xsd:any minOccurs="0" maxOccurs="unbounded" processContents="skip" />
      </xsd:sequence>
 </xsd:complexType>

</xsd:schema>
```

| Element | Description | Attributes and subelements |
|---|---|---|
| `templateId` | Identifier of the template to be used for exporting the sampled packets. | None |
| `interface` | Interface where packets are captured (observation point). | None |
| `selectAll` | Select all packets. | None |
| `countBased` | Apply count-based sampling. | `interval:` number of packets sample consecutively<br>`spacing:` number of packets that pass unsampled between two sampling intervals |
| `timeBased` | Apply time-based sampling. | `interval:` time interval in microseconds in which packets are sampled<br>`spacing:` time interval in microseconds between two sampling intervals |
| `randOutOfN` | Apply random n-out-of-N sampling. | `population:` number of elements in the parent population<br>`size:` number of elements taken from the parent population |
| `uniProb` | Apply uniform probabilistic sampling. | `probability:` this value divided by 4294967295 is the sampling probability |
| `nonUniProb` | Apply non-uniform probabilistic sampling. | not specified |

|  |  |  |
|---|---|---|
| `flowState` | Apply flow state sampling. | not specified |
| `filters` | Filters that restrict the considered flows. | `ipFilter` or alternative filter |

### 1.5 Exporting Process Types Definition (exporter-types-xml-schema.xsd)

Defines :

- exporter_type
- collector_type

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
       targetNamespace="diadem-firewall.org/MonitoringAPI"
       xmlns="diadem-firewall.org/MonitoringAPI"
       elementFormDefault="qualified">

 <xsd:annotation>
       <xsd:documentation xml:lang="en">
        XML Schema of types for exporting process configuration
       </xsd:documentation>
 </xsd:annotation>

 <xsd:complexType name="exporter_type">
       <xsd:sequence minOccurs="0">
        <xsd:element name="collectors" minOccurs="0">
              <xsd:complexType>
               <xsd:sequence>
                     <xsd:element name="collector" type="collector_type"
maxOccurs="unbounded" />
               </xsd:sequence>
              </xsd:complexType>
        </xsd:element>
        <xsd:element name="meterIds" minOccurs="0">
              <xsd:simpleType>
               <xsd:list itemType="xsd:unsignedInt" />
              </xsd:simpleType>
        </xsd:element>
        <xsd:element name="samplerIds" minOccurs="0">
              <xsd:simpleType>
               <xsd:list itemType="xsd:unsignedInt" />
              </xsd:simpleType>
        </xsd:element>
        <xsd:element name="aggregatorIds" minOccurs="0">
              <xsd:simpleType>
               <xsd:list itemType="xsd:unsignedInt" />
              </xsd:simpleType>
        </xsd:element>
        <xsd:element name="dataSendTimeout" type="xsd:unsignedInt" minOccurs="0" />
        <xsd:element name="templateTimeout" type="xsd:unsignedInt" minOccurs="0" />
        <xsd:element name="templateRefreshRate" type="xsd:unsignedInt" minOccurs="0"
/>
       </xsd:sequence>
       <xsd:attribute name="id" type="xsd:unsignedInt" use="required" />
       <xsd:attribute name="operation" type="operation_type" use="optional" />
 </xsd:complexType>

 <xsd:complexType name="collector_type">
       <xsd:sequence minOccurs="0">
        <xsd:element name="address" type="xsd:string" />
        <xsd:element name="port" type="xsd:unsignedInt" />
        <xsd:element name="protocol">
              <xsd:simpleType>
               <xsd:restriction base="xsd:string">
                     <xsd:enumeration value="udp" />
                     <xsd:enumeration value="tcp" />
                     <xsd:enumeration value="sctp" />
              </xsd:restriction>
             </xsd:simpleType>
        </xsd:element>
       </xsd:sequence>
       <xsd:attribute name="id" type="xsd:unsignedInt" use="required" />
       <xsd:attribute name="operation" type="operation_type" use="optional" />
  </xsd:complexType>

</xsd:schema>
```

| Element | Description | Attributes and subelements |
|---|---|---|
| `collector` | Collector the monitoring data is sent to. | `id:` collector identifier<br>`operation:` used in <edit-config><br>`address, port:` collector IP address and port<br>`protocol:` transport protocol |
| `aggregatorId, meterIds, samplerIds` | Aggregator identifier or identifer list of associated meters or samplers. | none |
| `dataSendTimeout` | In order to reduce the number of IPFIX messages, it is advantageous not to sent exported data sets immediately, but to wait until the IPFIX message is filled with a certain amount of data. This parameter indicates the maximum time (in microseconds) to wait for more data until an exported data set is sent. | none |
| `templateTimeout` | Time interval (in minutes) for periodical resending of the used templates to the collectors. If set 0, no time interval is used. | none |
| `templateRefreshRate` | Amount of IPFIX packets after which the used templates are resent. If set 0, template resending does not dependent on the amount of packets. | none |