

Project Number : IST-1999-10561-FAIN

Project Title : Future Active IP Networks



D7-Final Active Node Architecture and Design

Editor : Spyros Denazis

Document No: WP3-HEL-055-D7-Int

Contribution File Name : WP3-HEL-055-D7-Int.doc

Version : 1.0

Company : HEL

Date : 13 May 2003

Distribution : WP3,WP4, WP5

Dissemination: CO

Copyright © 2000-2003 FAIN Consortium

The FAIN Consortium consists of:

Partner	Status	Country
UCL	Partner	United Kingdom
JSIS	Associate Partner to UCL	Slovenia
NTUA	Associate Partner to UCL	Greece
UPC	Associate Partner to UCL	Spain
DT	Partner	Germany
FT	Partner	France
HEL	Partner	United Kingdom
HIT	Partner	Japan
SAG	Partner	Germany
ETH	Partner	Switzerland
FHG/ FOKUS	Partner	Germany
IKV	Associate Partner to FHG/FOKUS	Germany
INT	Associate Partner to FHG/FOKUS	Spain
UPEN	Partner	USA

The FAIN Consortium

University College London	(UCL)
Josef Stefan Institute	(JSIS)
National Technical University of Athens	(NTUA)
Universitat Politecnica De Catalunya	(UPC)
T-Nova Deutsche Telekom Berkom GmbH	(DT)
France Télécom / R&D	(FT)
Hitachi Europe Ltd.	(HEL)
Hitachi Ltd.	(HIT)
Siemens AG	(SAG)
Eidgenössische Technische Hochschule Zürich	(ETH)
Fraunhofer-Gesellschaft zur Förderung der angewandten Forschung e.V.	(FHG/FOKUS)
IKV++ GmbH Informations- und Kommunikationstechnologie	(IKV)
Integracion Y Sistemas De Medida, SA	(INT)
University of Pennsylvania	(UPEN)

Project Management

Alex Galis
University College London
Department of Electronic and Electrical Engineering,
Torrington Place
London WC1E 7JE
United Kingdom
Tel +44 (0) 207 458 5738
Fax +44 (0) 207 388 9325
E-mail: a.galis@ee.ucl.ac.uk

Authors

Spyros Denazis (HEL) – Editor
Toshiaki Suzuki (HEL) – Contributor
Chiho Kitahara (HIT) – Contributor
Thomas Becker (FHG/FOKUS) - Contributor
Lukas Ruf (ETH) – Contributor
Cornel Klein (SAG) – Contributor
Antonis Lazanakis (NTUA) – Contributor
Lawrence Cheng (UCL) – Contributor
Dusan Gabrijelcic (JSIS) – Contributor
Walter Eaves (UCL) - Contributor

Change History

Ver.	Date	Authors	Comments
0.1	11.03.2003	Spyros Denazis (HEL)	Initial ToC
0.2	10.04.2003	Spyros Denazis (HEL) Toshiaki Suzuki (HEL) Chiho Kitahara (HIT) Thomas Becker (FhG) Antonis Lazanakis (NTUA) Lukas Ruf (ETH) Cornel Klein (SAG) Lawrence Cheng (UCL)	First Draft of the following sections is added: <ul style="list-style-type: none">• Virtual Environments and Management (FhG)• RCF (NTUA, HIT)• Demultiplexing (HEL)• PromethOS (ETH, SAG)• Active SNMP Activator (UCL)• Wireless LAN Scenario was deleted as it was transferred to D9
0.3	8.05.2003	Spyros Denazis (HEL) Toshiaki Suzuki (HEL) Chiho Kitahara (HIT) Thomas Becker (FhG) Antonis Lazanakis (NTUA) Lukas Ruf (ETH) Cornel Klein (SAG) Lawrence Cheng (UCL) Dusan Gabrijelcic (JSIS)	Final versions of the following sections have been received. D7 is now ready to be submitted for review by the consortium. <ul style="list-style-type: none">• Virtual Environments Management & Java EE (FhG)• RCF (NTUA, HIT)• Demultiplexing (HEL)• PromethOS (ETH, SAG)• Active SNMP Activator (UCL)• Security architecture (JSIS)
0.4	9.05.2003	Lawrence Cheng (UCL)	Revisions of Active SNMP
0.5	9.05.2003	Antonis Lazanakis (NTUA) Spyros Denazis (HEL)	<ul style="list-style-type: none">• RCF Conclusions added and other editorial changes• Minor editorial changes (formatting)
0.6	11.05.2003	Dusan Gabrijelcic(JSIS)	<ul style="list-style-type: none">• Security updated
0.7	12.05.2003	Spyros Denazis (HEL) Toshiaki Suzuki (HEL) Thomas Becker (FhG)	<ul style="list-style-type: none">• Executive Summary and Conclusions (HEL)• Demultiplexing Minor modifications (HEL)• Virtual Environments Management & Java EE Updated Version (FhG)
0.9	13.05.2003	Alex Galis (UCL)	Overall review and modifications
0.91	13.05.2003	Lawrence Cheng (UCL)	Includes contributions from Alex, Antonis, Dusan, Lukas and myself.
1.0	13.05.2003	Spyros Denazis (HEL)	Final Modification and completion of the editorial work

Executive Summary

This deliverable (D7) is the third and final in a series of deliverables (D3, D5 and D7) that described the architecture and implementation of the FAIN Active Node. These annual project deliverables were produced as phased revisions: Concept revisions and /or Technical revisions.

Concept revisions refer to the main architectural concepts outlined in year 1 or year 2 deliverables as they needed more focus in some cases or lacked completeness in the previous version of the deliverable. In the year 3 deliverable we have revisited the concepts and described them from a different viewpoint while making the necessary references to the corresponding implementation, thereby adding more depth in their description by connecting them with experimental proofs.

Technical revisions refer to the implementation of the FAIN Active Network architecture, which resulted in modifications, or extensions of the initial version of the architecture description as well as the particular choice of technologies and the engineering aspects thereof.

A full overview of the FAIN project architectural results are described in deliverable D14. As a result what is described here are the design and implementation details of the FAIN node components mentioned in the D14 common part.

Section 1, presents a short introduction to the FAIN Active Node whereas details may be found in D14. Section 2, describes the Virtual Environment Management (VEM) framework and how it has been designed to realise the component-based feature of the FAIN node. Around VEM, the other components of the node have been designed and integrated. More specifically, Resource Control Framework (RCF), Demultiplexing, and Security are presented in Sections 3, 4, and 5. Section 6 consists of the description of different Execution Environments that have been identified, designed and implemented. In Section 6.1, we describe a Java EE which is in fact the implementation of the VEM framework and through the offered management functionality binds together all the FAIN node components as well as the other EEs. Section 6.2, describes the architecture and functionality of a high performance EE which also supports the deployment of new components in the same way as the Java EE and entirely resides in the transport plane. Section 6.3 describes an Active SNMP EE based on the SNAP EE which resides in the control plane and is used in order to control and configure the transport plane and in particular a router. In the last Section of this deliverable we provide our conclusions. More elaborate conclusions are presented in the new deliverable D14 where the overall achievements of the project are provided.

TABLE OF CONTENTS

1	FAIN OVERVIEW	9
2	VIRTUAL ENVIRONMENTS & MANAGEMENT	10
2.1	INTRODUCTION.....	10
2.2	REQUIREMENTS.....	11
2.3	DESIGN.....	11
2.3.1	<i>Basic Component</i>	12
2.3.2	<i>Configurable Component</i>	12
2.3.3	<i>Component Manager</i>	13
2.3.4	<i>Template Manager</i>	13
2.3.5	<i>Resource Manager</i>	14
2.3.6	<i>Special Managers</i>	14
2.4	CONCLUSION	15
3	RESOURCE CONTROL FRAMEWORK.....	16
3.1	INTRODUCTION.....	16
3.2	REQUIREMENTS.....	16
3.2.1	<i>RCF Design</i>	16
3.3	RCF MAIN FUNCTIONALITIES	17
3.3.1	<i>Admission Control</i>	17
3.3.2	<i>Resource Control</i>	19
3.4	MODEL RCF IMPLEMENTATION	20
3.4.1	<i>Traffic Control and Management for Linux</i>	21
3.4.2	<i>DiffServ Control and Management for a Gigabit Router</i>	22
3.5	CONCLUSIONS	23
4	DEMULPLEXING.....	24
4.1	REQUIREMENT FOR DEMULPLEXING	24
4.1.1	<i>Requirement for Active Packet format for Demultiplexing</i>	24
4.1.2	<i>Requirement for Demultiplexing Mechanism</i>	24
4.2	DEMULPLEXING FRAMEWORK.....	24
4.2.1	<i>Active Channel</i>	25
4.2.2	<i>Data Channel</i>	28
4.3	CONCLUSION ON DEMULPLEXING	29
5	SECURITY	30
5.1	INTRODUCTION.....	30
5.2	SYSTEM RELATIONSHIPS AND ENTITIES	30
5.3	THREATS, SECURITY REQUIREMENTS AND ARCHITECTURE GOALS	32
5.4	SECURITY ISSUES.....	33
5.4.1	<i>Authorization and policy enforcement</i>	33
5.4.2	<i>Authentication</i>	34
5.4.3	<i>Packet integrity</i>	34
5.4.4	<i>System integrity</i>	35
5.4.5	<i>Code and service verification</i>	35
5.4.6	<i>Limiting resource usage</i>	35
5.4.7	<i>Accountability</i>	36
5.5	HIGH LEVEL SECURITY ARCHITECTURE	36
5.6	FAIN ARCHITECTURAL MODEL AND SECURITY ARCHITECTURE	36
5.7	SECURITY ARCHITECTURE DESIGN AND IMPLEMENTATION.....	38
5.7.1	<i>Building components security context</i>	38
5.7.2	<i>Enforcement layer, authorization and policy enforcement</i>	39
5.7.3	<i>External security representation</i>	39
5.7.4	<i>Cryptographic subsystem and secure store</i>	40
5.7.5	<i>Connection manager</i>	40
5.7.6	<i>Verification manager</i>	40
5.8	GENERAL ACTIVE PACKET SECURITY EVENTS.....	41

5.9	SECURITY ARCHITECTURE PERFORMANCE.....	41
5.10	ARCHITECTURE APPLICABILITY.....	43
5.11	EVALUATION OF THE SECURITY ARCHITECTURE.....	44
5.12	CONCLUSIONS.....	45
6	EXECUTION ENVIRONMENTS.....	46
6.1	JAVA EE.....	46
6.1.1	<i>Introduction.....</i>	46
6.1.2	<i>Implementation.....</i>	46
6.1.3	<i>Use Cases.....</i>	50
6.1.4	<i>Conclusion.....</i>	51
6.2	PROMETHOS EE.....	52
6.2.1	<i>Architectural Overview.....</i>	53
6.2.2	<i>Netfilter Framework.....</i>	53
6.2.3	<i>PromethOS Netfilter-Table.....</i>	55
6.2.4	<i>Plugin Framework and Execution Environment.....</i>	56
6.2.5	<i>PromethOS User Space Library.....</i>	56
6.2.6	<i>Summary, Outlook and further work.....</i>	57
6.3	ACTIVE SNMP ACTIVATOR.....	58
6.3.1	<i>Introduction to SNAP EE.....</i>	58
6.3.2	<i>System Design Goals.....</i>	58
6.3.3	<i>System Design.....</i>	59
6.3.4	<i>Introduction to the ANEP-SNAP Packet Engine (ASPE).....</i>	62
6.3.5	<i>Requirements of the ASPE.....</i>	63
6.3.6	<i>ANEP-SNAP Packet Format.....</i>	64
6.3.7	<i>System Architecture.....</i>	65
6.3.8	<i>Conclusion.....</i>	66
7	CONCLUSIONS.....	67
8	REFERENCES.....	68
A.1	INTERFACE DEFINITIONS OF THE JAVA EXECUTION ENVIRONMENT.....	71
A.1.1	<i>Identification.....</i>	71
A.1.2	<i>Properties.....</i>	71
A.1.3	<i>Ports.....</i>	71
A.1.4	<i>Interface iComponentInitial.....</i>	72
A.1.5	<i>Interface iConfiguration.....</i>	73
A.1.6	<i>Interface iConfigurationObserver.....</i>	74
A.1.7	<i>Interface iTemplateManager.....</i>	74
A.1.8	<i>Interface iComponentManager.....</i>	75
A.1.9	<i>Interface iResourceManager.....</i>	76
A.1.10	<i>Interface iResourceMonitor.....</i>	77
A.1.11	<i>Interface iResourceObserver.....</i>	77
A.1.12	<i>Interface iMonitoredResourceManager.....</i>	78
A.1.13	<i>Interface iVirtualEnvironmentManager.....</i>	78
A.2	IMPLEMENTATION OF PROMETHOS.....	78
A.2.1	<i>PromethOS Netfilter-Table: iptables_promethos.c.....</i>	78
A.2.2	<i>PromethOS Netfilter-Target: ipt_PROMETHOS.c.....</i>	80
A.2.3	<i>Plugin Implementation.....</i>	81
A.2.3.1	<i>load().....</i>	81
A.2.3.2	<i>unload ().....</i>	81
A.2.3.3	<i>target().....</i>	81
A.2.3.4	<i>config().....</i>	81
A.2.3.5	<i>reconfig().....</i>	82
A.2.3.6	<i>print().....</i>	82
A.2.4	<i>Example Plugin Explained.....</i>	83
A.2.5	<i>The PromethOS User Space Library.....</i>	84
A.2.6	<i>Example Use of PromethOS Plugin Framework.....</i>	88
A.2.7	<i>Example Use of the PromethOS User Space Library.....</i>	89

Table of Figures

FIGURE 1-1: OVERVIEW OF FAIN NODES	9
FIGURE 2-1: AN EXAMPLE SERVICE CONSISTING OF SEVERAL COMPONENTS.	11
FIGURE 2-2: HIERARCHY OF COMPONENT ABSTRACTIONS.	12
FIGURE 2-3: INITIAL SETUP OF THE MANAGEMENT LAYER OF A FAIN ACTIVE NODE	15
FIGURE 3-1: RCF ARCHITECTURE	17
FIGURE 3-2: ADMISSION CONTROL. INVOLVED COMPONENTS HIERARCHY	18
FIGURE 3-3: MULTILEVEL HIERARCHICAL RESOURCE SHARING	19
FIGURE 3-4: RESOURCE CONTROL. COMPONENTS AND INTERFACES	20
FIGURE 3-5: TRAFFIC CONTROL AND MANAGEMENT FOR LINUX SOFTWARE ROUTER	22
FIGURE 3-6: DIFFSERV CONTROLLER FOR GIGABIT ROUTER	23
FIGURE 4-1: BLOCK DIAGRAM OF PACKET DELIVERY	25
FIGURE 4-2: ACTIVE PACKET TRANSMISSION	26
FIGURE 4-3: ANEP PACKET FORMAT	27
FIGURE 4-4: VIRTUAL ENVIRONMENT IDENTIFIER	28
FIGURE 4-5: EXECUTION ENVIRONMENT IDENTIFIER	28
FIGURE 4-6: DATA PACKET TRANSMISSION	29
FIGURE 5-1: SYSTEM ENTITIES AND RELATIONSHIPS	31
FIGURE 5-2: HIGH LEVEL SECURITY ARCHITECTURE	36
FIGURE 5-3: SECURITY CONTEXT, VE AND SERVICE STARTUP	38
FIGURE 5-4: SECURITY RELATED PACKET COSTS	42
FIGURE 6-1: CLASS HIERARCHY FOR JAVA EXECUTION ENVIRONMENT	47
FIGURE 6-2: NETFILTER AND PROMETHOS	53
FIGURE 6-3: NETFILTER ARCHITECTURE FOR IPV4	54
FIGURE 6-4: PROMETHOS NETFILTER-TABLE HOOKS	55
FIGURE 6-5: SNAP PACKET FORMAT	59
FIGURE 6-6: SNAP ACTIVATOR BLOCK DIAGRAM	61
FIGURE 6-7: ANEP-SNAP PACKET FORMAT	64
FIGURE 6-8: BLOCK DIAGRAM FOR ANEP-SNAP PACKET FLOW	65

Table of Tables

TABLE 4-1: DATABASE FOR THE ACTIVE PACKET	26
TABLE 4-2: DATABASE FOR THE DATA PACKET	29
TABLE 3: EXAMPLE CODE OF THE PROMETHOS USER SPACE LIBRARY	90

1 FAIN OVERVIEW

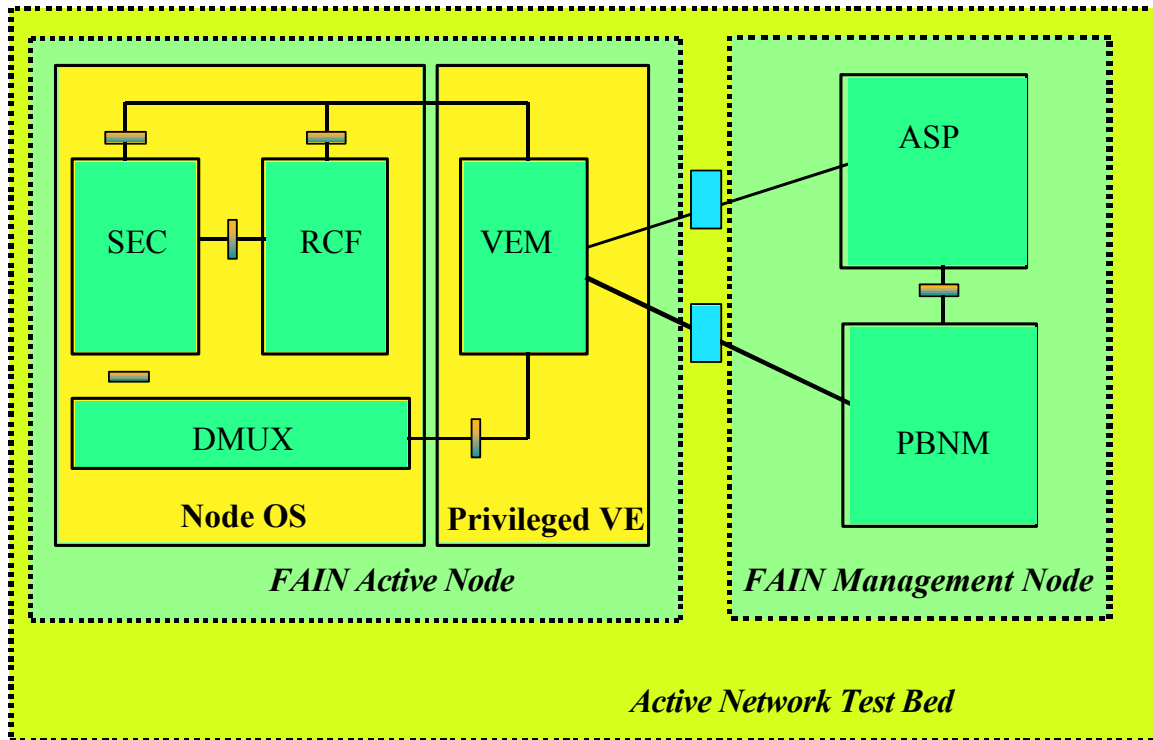


Figure 1-1: Overview of FAIN Nodes

Figure 1-11 provides an overview of the major AN node components and their corresponding interfaces that comprise the FAIN AN node architecture.

The main components of the FAIN Active Node are presented in the following chapters and they are:

- Virtual Environment Management (VEM)
- Resource Control Framework (RCF)
- Demultiplexing
- Security
- EE Types: Java EE, High-performance EE, SNAP EE

2 VIRTUAL ENVIRONMENTS & MANAGEMENT

2.1 INTRODUCTION

The concept of virtual environments was introduced in FAIN to overcome the problem of having several execution environments implemented in various technologies and providing different abstractions, interfaces, etc. For example, in FAIN we implemented three kinds of execution environments, based on JAVA/CORBA, SNAP, and PromethOS (see chapter 6).

While execution environments support the installation, instantiation, and configuration of active services' code in various ways, the virtual environment puts a uniform management layer on top. This allows external clients to interact with services through the interface of the virtual environment in a generic way and the interactions will be mapped to specific interfaces of the execution environments.

Several execution environments can be attached to a virtual environment just in the same way as other resources. This leads to another aspect of virtual environments, which is the partitioning of resources. As defined in the FAIN business model [9], there may be a number of service providers acting as the customers of a network provider. The network provider can set up virtual environments on selected network nodes and assign them to a particular service provider in order to offer a virtual active network. The access to the virtual environments will be made available to the respective service provider so that it can manage its own virtual network. The resource partitioning implemented among virtual environments will prevent interferences with other service providers and additionally allow an accounting per service provider.

A special virtual environment, also called the privileged virtual environment, will be started when an active node is booted. This environment belongs to the network provider and contains the fundamental services such as management of basic resources (CPU, memory, bandwidth) as well as management of virtual environments plus different kinds of execution environments. The privileged virtual environment also provides a means for the network provider to manage the nodes inside an active network.

Several virtual environments belonging to the same service provider but running on different active network nodes will form a virtual network to be used by the service provider to deploy services and make them available to customers. In order to know which virtual environments belong to a particular virtual network the environments get tagged with a special network identifier.

To summarise, the concept of virtual environments enables several aspects:

- a generic way of deploying and managing active services independent of the technology of the underlying execution environment,
- a generic way to manage (i.e. monitor and control) active nodes for service providers as well as for network providers,
- the partitioning of resources among several service providers,
- the accounting of resource usage per service provider, and
- the delegation of service management to the service providers.

In the following the requirements towards the node level management and its design are presented in more detail.

2.2 REQUIREMENTS

The major requirement towards the design of the interface of a virtual environment is to provide a generic way of deploying and managing services on active network nodes offering different execution environments. Further, the deployment and management should be possible in a flexible and dynamic way of facilitating creation, configuration, and reconfiguration of services on demand. Interaction between services should be supported in a safe and controlled way. Nonetheless, the development of services should be easy and concentrated on the service logic by avoiding to re-implemented commonly needed functionality. Last but not least, the implementation of the node level management layer (i.e. the virtual environments) should be portable between different operating systems and should interoperate with external components potentially implemented in different programming languages.

In the following we will show how those requirements were met.

2.3 DESIGN

In FAIN, services are described in a component-based approach. This approach allows the creation of new services by combining already available components and potentially configuring them. However, any missing functionality has to be implemented somehow. Consequently, a component-based approach has been chosen for the deployment and management of services as well as for the development of services. Legacy systems missing a component-oriented interface can be wrapped by components residing in the node management layer (see figure 1).

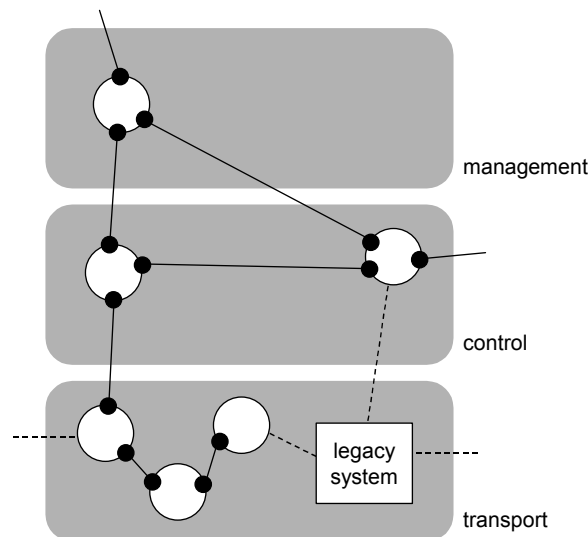


Figure 2-1: An example service consisting of several components.

A component-based runtime environment for services makes it easier to develop services. Aspects such as lifecycle management, configuration, access control, monitoring, etc. can be implemented by the supporting framework. Thus the developer of a service can concentrate on the service logic and frequently needed aspects don't have to be implemented over and over again.

By providing means to dynamically inter-connect components a high degree of flexibility can be reached. After the initial setup services may be reconfigured during runtime by adding new components, removing components, or changing connections between components. This applies not only to high-level services but also to the basic service offered by an active node. For example, a specific service component may be connected to a "channel" component providing the dispatching of packets belonging to a particular flow while also controlling the bandwidth.

Components are created and destroyed by component managers. A component manager is responsible for one type of component and besides the creation and deletion it offers methods for activating, deactivating, and finding component instances. Thus a component manager implements the so called factory and finder patterns.

In order to deploy a service, which needs components that are not already available on a particular network node, the respective component managers have to be loaded into an appropriate execution environment. Since a component manager functions as a template from which instances can be created the execution environments are template managers allowing to install, uninstall, and find component managers. As virtual environments provide an abstraction from particular execution environments they also need to implement the template manager pattern.

Specialised component managers are used to manage specific resources. Additionally to the factory and finder patterns a resource manager offers methods for monitoring instances with regards to certain dimensions like usage of memory usage or CPU cycles.

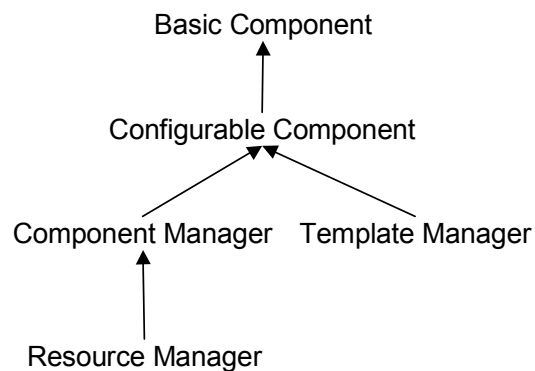


Figure 2-2: Hierarchy of component abstractions.

Figure 2-2 depicts the hierarchy of abstractions used for service components. In the following sections these abstractions will be presented in more detail. The implementation of the abstractions makes up the runtime support framework for service components and is described in chapter 6.1.

2.3.1 Basic Component

The objective for the design of the FAIN component model has been its generality. It should be possible to cover all other models with this one. Thus, the FAIN model is quite simple: a basic component has a defined owner, a unique identifier, and optionally offers a couple of ports through which its specific functionality could be accessed.

A port will have a particular format and address so that it can be accessed from the outside. The values of the format and address are expressed as arbitrary character strings, which are transparent to the service runtime framework but have to be understood by the component itself and its communication peers. At least one port is offered by all components, which is the initial port. This port is used by clients to query other supported ports and to get access to them. In order to get access to a port a client must authenticate himself.

2.3.2 Configurable Component

A configurable component is derived from a basic component and additionally offers a configuration port. This port is used to get and set the configuration of the component in the form of properties, i.e. pairs of names and values. Interested clients can connect a call-back port to receive notifications when selected properties change their values.

Further, the configuration port allows connecting the ports of the respective component with ports of other components. This is used when a service is deployed and components have to be interconnected as described in chapter 2.3

2.3.3 Component Manager

A component manager is derived from a configurable component. It offers a port for managing component instances comprising the following:

- Creation of instances with specifying a profile: the component manager will create a new instance in a standby mode and store it together with the profile. The result is a unique identifier for the new instance. If no activation occurs for the new instance within a specific timeframe the instance will be deleted automatically.
- Activation of instances with specifying initial setup parameters: the component manager will put the new instance into action and initialise it with the setup parameters. The new instance is now ready to interact with its environment.
- Deactivation of instances: the component manager will put the instance back into the standby mode. After a specific timeframe the instance has to be activated again or it will be deleted.
- Deletion of instances: the component manager will simply delete the instance.
- Discovery of instances: the component manager will return a reference to a desired instance or a set of instances based on several criteria, e.g. the instance's unique identifier, an owner, or other properties.

2.3.4 Template Manager

A template manager is derived from a configurable component. It offers a port for managing templates, where a template corresponds to a particular component manager instance. There are exactly two occurrences of template managers: execution environments and virtual environments. While execution environments are responsible for putting component managers into action the task of a virtual environment is to dispatch requests to an appropriate execution environment. The role of the privileged virtual environment is to dispatch requests to the virtual environment owned by the appropriate service provider and can thus be used as the initial point of contact for any client.

Managing templates comprises:

- Installation of templates with specifying a template description: the environment will take the required steps to put the corresponding component manager in action. An installation request will eventually arrive at the appropriate execution environment. The execution environment will use specific means for installing a component manager depending on the underlying technology, e.g. instantiating a new JAVA class loader or copying object files to appropriate locations. The template description includes a name, a version, the identifiers of the target virtual and execution environments, the path to the templates code base, and the entry point for starting the corresponding component manager. The result of the installation is a unique identifier for the new template.
- De-installation of templates with specifying the template's unique identifier: the environment will delete the corresponding component manager. It is specific to the template – and thus part of the implementation of the component manager – whether running component instances should be deleted, too.
- Discovery of templates: the environment will return a reference to a component manager or a set of component managers according to various criteria, e.g. the template's unique identifier, its name, version, or identifiers of environments.

2.3.5 Resource Manager

A resource manager is derived from a component manager. It offers a port for monitoring component instances and for registering call-back ports to get notifications when certain thresholds are reached.

The methods offered by a component manager are extended in the following way:

- Creation of instances with specifying a resource profile: the resource manager can use this profile for checking the availability of resources needed for putting the new instance into action. The required resources will be kept in a standby mode for a certain amount of time in order to be usable by the new instance when it gets activated. If no activation occurs within the timeframe the resources will be released.
- Activation of instances with specifying initial setup parameters: the new instance is now ready to use the assigned resources and the resources are bound to the instance.
- Deactivation of instances: the resource manager will put the resources assigned to the instance back into the standby mode. After a specific timeframe the instance has to be activated again or the resources will be freed.
- Deletion of instances: the resource manager will free all resources assigned to the instance.
- Registering a call-back: interested clients can register a call-back port in order to receive notifications when the usage of resources by an instance reaches particular limits. This can be done for upper or lower limits.

2.3.6 Special Managers

During the boot procedure of a FAIN active node the privileged virtual environment is started together with an attached execution environment. When a new virtual environment is created it will need some basic resources in order to support template installation and component instantiation. For this reason various resource managers are installed inside the privileged virtual environment during the boot procedure.

Those basic resources managers comprise:

- A virtual environment manager for the creation of new virtual environments. This manager will examine the resource profile and try to create any referenced resource using the other basic managers. The resulting resource components will be attached to the new virtual environment.
- A number of execution environment managers for the creation of specific execution environments. Since all templates have to be installed in an execution environment and running instances can also exist only inside execution environments, there has to be at least one execution environment attached to a virtual environment. Specific execution environments will be described in chapter 6.
- A security manager for creating security contexts. A security context will hold information about the identity and security policies of the owner of an environment, i.e. the network provider for the privileged virtual environment or a service provider for other virtual environments. The security context is used to check interactions with components belonging to the respective environment.
- A channel manager for creating channels. Component instances running in an execution environment can connect to a channel to receive and send packets from and to the network. The channel manager is responsible for dispatching packets to the appropriate channels.
- A traffic manager for creating traffic controllers. A traffic controller can be used by component instances to control particular packet flows. For example, a traffic controller may offer methods for setting up a guaranteed bandwidth or a specific packet scheduling.

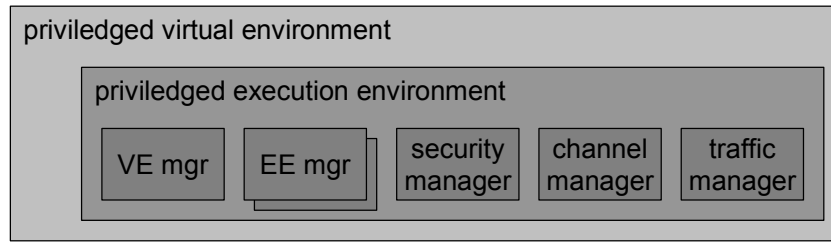


Figure 2-3: Initial setup of the management layer of a FAIN active node.

Figure 2-3 shows the initial setup of the management layer of an active node. The privileged execution environment runs in the context of the privileged virtual environment. Inside the privileged execution environment there are the resource managers for the basic services. They will be used to create resources for other virtual environments. Details of the implementation are described in chapter 6.1.

2.4 CONCLUSION

This chapter presented the management layer on the node level. The introduction of virtual environments allowed integrating several execution environments with potential different implementation technologies. Further, physical resources could be partitioned among several node users – the service providers – with the help of virtual environments. To achieve a flexible and fine grained control over service deployment and management a component-based approach was chosen for the node level management layer. With the introduction of properties and ports for components a means was found for dynamic reconfiguration of services in that service components' properties and interconnections between service components could be changed during the service's runtime.

3 RESOURCE CONTROL FRAMEWORK

3.1 INTRODUCTION

In this section the Resource Control Framework (RCF) of the FAIN nodes is described. RCF is the part of the virtual environment management (VEM) framework that is involved in the management of resources and other node components and performs the runtime control of resources inside the FAIN ANs. The importance of RCF is considered as very high as it supports one of the major concepts of the FAIN project: the bounding of Virtual Environments (VEs) in the FAIN ANs with specific resource capacity. RCF supports this bounding, isolates various VEs in the same node and supports dynamically the lifecycle of each VE by allocating, controlling and releasing resources correspondingly.

3.2 REQUIREMENTS

The main objective of Active Networking of FAIN is to provide the necessary infrastructure for the dynamic deployment of the new services and to give the user the ability to customise the network using his own code. This dynamic aspect of active networks has an impact on the requirements for resource management of active node. In comparison to a traditional network node, active node offers the ability to dynamically inject code, which implements a new service. The code is injected in an execution environment that belongs to a VE, which presents an abstraction of the node to the active application. In order to be able to support these services the VE should have guaranteed access to the necessary resources of the node. Hence, RCF should provide the necessary mechanisms to enforce resource sharing among the various users. Every resource allotment in the form of a VE should be isolated and independent from other VEs. Furthermore, the dynamic creation, deletion and reconfiguration of VEs should be supported at any time that the status of the resources does not forbid it. An other important rule that should be followed during the control of the FAIN AN resources is that the admission of the creation of a new VE should not affect the contracts and the agreements of the existing VEs. In addition, not only should RCF provide access to the allocated resources, but also a set of the necessary control and management mechanisms, in order the VE owner to be able to use and manage these resources according to his desire and necessities. Finally, an open interface that gives the necessary access to RCF functionality of the VE should be provided to its owner.

3.2.1 RCF Design

In Figure 3-1, the design of RCF is depicted. For every resource that is controlled a resource controller (RC) takes over its runtime control and a resource manager (RM) to manage the partition of the resource among the VEs. Finally for every VE exists a resource controller abstraction (RCA) that represent part of the RC functionality, specifically for the part of the resource that have been allocated to the VE, to the VE client.

In more details the main categories of RCF components are the following:

Resource Controller (RC): RC is an entity, which is responsible for the actual control of a resource inside the FAIN Active Node. RC can be a component running in Kernel Space of the node for a software router or can be a specific device of a hardware router. Moreover, a RC can vary from a simple scheduler (e.g. CPU Scheduler) to a more complex framework, which could control a whole mechanism in the kernel that includes the control of more than one real resource, e.g. for Linux based AN can be the Netfilter Framework or the Traffic Control Framework. Every RC has an interface that allows its runtime configuration, which includes the allocation and monitoring of the resource for which is responsible for.

Resource Manager (RM): For every RC, a RM exists in user space. It is responsible for the configuration of the corresponding RC in order to enforce the resource partition among the various VEs. Moreover, RMs are responsible for the RCAs creation, configuration and management. Among others, the RMs are responsible for the Admission Control of the incoming requests for new allocations and for the realization of the allocation by configuring the corresponding RC.

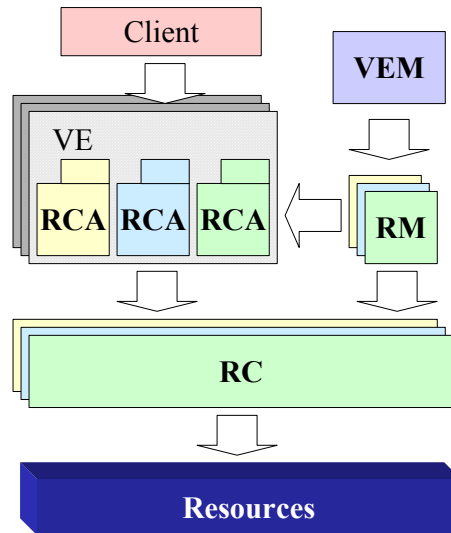


Figure 3-1: RCF Architecture

Resource Controller Abstraction (RCA): For every resource that is allocated to a VE, a RCA of that resource is created. The RCA represent the part the RC that controls the allotted to the VE resource and it is bounded up with a specific part of the whole resource. The RCAs export IFs and accept requests by VE owners for resource access. Resource access includes requests for resource consumption and management. RCAs check those requests against resource status and VE owners' privileges and enforce the valid requests by configuring the corresponding RCs accordingly.

The RMs and RCAs are part of the VEM framework as RMs are specific component managers and RCAs are specific configurable components. Virtual Environment Manager (VEM) is the higher manager in hierarchy and manages all the RMs. In a sense, VEM is also a RM that manages the high level resource that called VE.

RCs generally are not part of the virtual environment management framework. Mostly are platform dependent entities and mechanisms that their mission is to control a specific or a set of resources. Each of them has a configuration interface which gives to the corresponding RM and RCAs the ability to dynamically determine the way that its control the resource by changing its configuration.

3.3 RCF MAIN FUNCTIONALITIES

In order RCF to meet its requirements acts in two different ways. The first is to make the admission control of any new VE creation request. The second is to control and manage the usage of the resources of the admitted VEs.

3.3.1 Admission Control

The creation of a new VE in the FAIN Node cannot always be accepted because of the finite amount of resources. This requires the existence of an Admission Control (AC) mechanism within the RCF of the FAIN Node.

AC in the FAIN Node addresses a set of actions that should be made by the RCF during the VE's creation phase (or during re-negotiation phase) in order to decide whether a VE creation request should be accepted or rejected.

A new VE can be admitted to the node only if its requirements for resources can be satisfied without at the same time any commitments that have been made to the existent VEs to be violated. The final decision for the acceptance or not of the request for the creation of a new VE the unreserved resources and the needs of the new VE. In parallel the increase of the node's utilization should be achieved by the acceptance of as many VEs as possible. In other words RCF should not refuse the creation of a VE that node status in terms of resources shows that can be admitted.

The AC decision for every resource is based on an algorithm, which is not necessary to be the same for every resource. The RCF framework does not define specific algorithms for specific resources as it is beyond its scope and can vary according to the resource. It just defines the mechanism, the involved components, the interactions between them, and above all the admission control decision point for the FAIN AN.

3.3.1.1 Admission Control Model

VE includes a set of different resources for which different RMs are responsible. Hence, AC isn't performed by a central object but by the various involved RMs independently. Of course, for the overall decision when the creation of a new VE can be admitted or not the responsible component is the VE Manager (VEM). The VEM in order to make the final decision gets in contact to the involved RMs and asks them if they can or cannot admit the new VE. The RMs perform AC independently and if the responses from all of them are positive the VEM's decision will be positive as well. Otherwise the VEM rejects the VE creation request.

Figure 3-2 depicts the relationships and the hierarchy between the involved components to the AC process. The VE creation requested entity is a client from RCF AC functionality perspective, hence, from now on will be called Client. According to FAIN business model [9], probably, Client cannot be someone/something else than a service provider (SP) or a SP's agent or the Network Management System on behalf of a SP. Of course, the client's nature does not affect in any way the AC process.

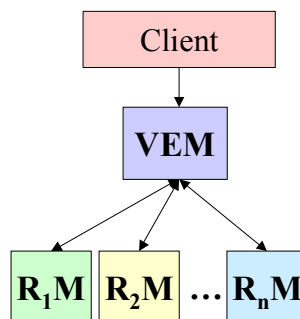


Figure 3-2: Admission Control. Involved Components Hierarchy

In FAIN we adapted a two-phase approach for AC, the creation phase and the activation phase.

During the creation phase, the client requests the creation of a new VE. The request includes a resource profile, which describes the client's requirements in resources. VEM receives the request and breaks it down into single resources requirements. Then, VEM passes these requirements to the corresponding RMs. Every RM decides if the needed allocation can be done or not, and if the reply is positive, pre-allocates the resource and replies positively to the VEM. The pre-allocation includes the creation of a RCA for that part of the resource, but in a standby mode. In addition, no configuration of the RC for the actual allocation of the resource is occurred. That means that the VE client cannot use the part of the resource, which is pre-allocated, but neither this part can be allocated by someone else. The resources remains pre-allocated until an activation or a withdraw request arrives, or until the expiration of a specific timeout. When the VEM has collected all the replies from the RMs, it can decide to admit or not the new VE. If any of the RM replies negatively, the VEM reply will be negative as well, and also it gets in contact with the rest RMs in order to inform them that the new VE

will not be created, and therefore they have to release the pre-allocated resources.

When all the replies from the RMs are positive, the VEM replies positively as well. But even then, the VEM does not activate the newly created VE and the resources remain preallocated. The reason is that generally every VE is part of a virtual private active network (VPAN). During the creation of a VPAN, different VE creation requests are made to different ANs. It is possible the request in one or more of the nodes to be rejected and therefore the creation of the VPAN not to be feasible. Then the admitted and created VEs should be removed without using them. On the other hand, the resources on every node that the VEs have been admitted should be shielded from anyone who simultaneously tries to set up a VPAN.

If all the creation of all the VEs of a VPAN has succeeded, the activation of each created VE required before being ready for use. On the node level, the activation request arrives at VEM. VEM gets in contact with all the involved RMs in order to activate the RCAs and configure the RCs accordingly, in order to enforce the allocation.

3.3.2 Resource Control

One of the main duties of RCF is the partitioning of resources to the various users and the real time control of their consumption. Every player who desires the use of the node should request the assignment of a part of the resources, in terms of specific capacity. This part of the node capacity is assigned to a VE. Moreover RCF is responsible to export interfaces to VEs' clients, which interfaces offer resource access and management capabilities to VEs' clients.

3.3.2.1 Resource Control Model

RCF supports a multilevel, hierarchical resource sharing, as it is shown in Figure 3-3. The first level of sharing is among the various VEs. RCF then exports a control I/Fs to every VE, one for every resource which have been allocated to the VE, that allows the further partitioning of the resources according to owners desire. For example in the Figure 3-3 the node is shared by three VEs. Then the VE₁ is partitioned in 3 shares, the VE₂ is not partitioned at all, and the VE₃ is partitioned in two shares, the one of which is further partition to

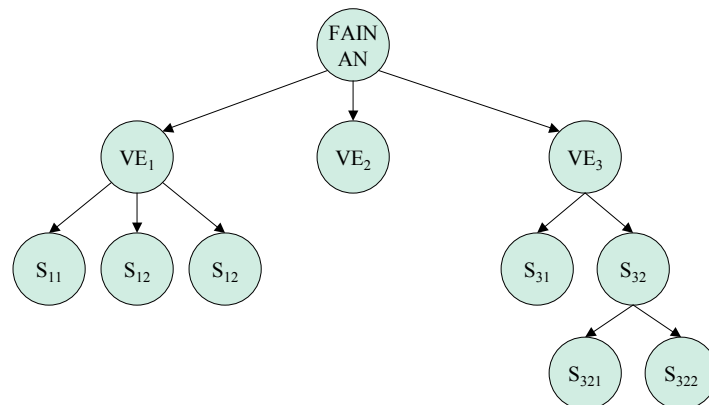


Figure 3-3: Multilevel Hierarchical Resource Sharing

RCF is responsible for the first partitioning among the VEs in the sense that does not define how the resources will be further partitioned. On the other hand RCF supplies VEs' owners with the appropriate mechanisms in order to be able to further partition their resources according to their will.

For the first level partitioning, namely the partition of the overall node resources among VEs, we use the hard allocation approach. Apart from being the simplest managed solution, there are three important reasons for that choice. For starters, according to FAIN business model [9], FAIN network is owned by Network Provider (NP). Multiple Service Providers (SPs) request to use part of network infrastructure in order to deploy their services. NP provides this infrastructure by setting up VEs to

multiple nodes. Every VE represents a part of node resources. In other words, FAIN aims to be a commercial network, where NP sells network infrastructure to SPs. Therefore, SP should be enforced to buy the infrastructure that wants to use, and not be able to consume any unused resources without to be charged. Secondly, SP is charged for the resources, which have been provided to him, and he demands to have 100% guarantee access to them. Hard allocation is the only method that ensures this guarantee. Finally, SP is responsible to manage his VEs, so he is able to use resource control techniques that lead to high utilization. Hence, if we have chosen to use an over-allocation technique, the result would have been that in many cases the requirements of SPs in terms of resources not to be satisfied. So hard allocation is considered the best choice, without the general design of RCF to exclude other approaches.

Apart from the allotted part of resources, NP provides SP with resource control mechanisms. These mechanisms allow the efficient management of the allotted resources by the SP. The SP has been given the capability to further partition the resources that belongs to him, choosing if it is desirable, to use soft allocation, or hard allocation or combination of the above. In addition SP can choose among a variety of resource sharing algorithms to partition its resources. Obviously, SP aims to utilize all the node capacity that he has charged for.

3.3.2.2 Components and Interfaces

Figure 3-4 depicts the involved components and interfaces and the interactions between them during the resource control and management process.

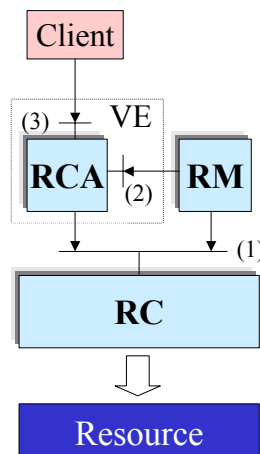


Figure 3-4: Resource Control. Components and Interfaces

The responsible component for the run-time control of every resource is the corresponding Resource Controller (RC). For every resource that is controlled exist a separate RC. Every RC has an interface (Figure 3-4 – (1)) that allows its configuration. The Resource Manager (RM) and Controllers (RCs) have access to that Interface. The RM uses that interface to partition the resource to the various VEs. For every VE allotment, the RM creates a Resource Controller Abstraction (RCA) for it. RM uses the configuration interface (Figure 3-4 – (2)) of the new RCA to configure it, in order to have access to the appropriate allotment of the resource. The RCA exports a control interface (Figure 3-4 – (3)) to VE client in order to be possible the use of the resource. In addition that interface allows in some extend the building of resource control mechanisms, that determine how this part of the resource will be used, without of course this to have any effect out of the VE. The last is ensured by the RCAs, which intersect and check every configuration request by VE clients. All the configuration requests from the Client to RCA, first are checked and then are forwarded to the corresponding RC.

3.4 MODEL RCF IMPLEMENTATION

The variety of the different resources that are deemed as essential to be controlled in an active node is very wide. They can vary from network resources, like the bandwidth of the links, to computational

resources, like the CPU cycles and memory space. From all these resources, we chose to control for the model RCF implementation the outbound bandwidth because bandwidth still is considered as the most valuable resource for every existing network architecture and because the Linux TC framework [16] existence. Linux TC exists as standard component in the kernel of every recent Linux distribution and it is a very powerful framework for the control of traffic.

3.4.1 Traffic Control and Management for Linux

Based on the general RCF framework and as part of virtual environment management (see chapter 2) implementation a Traffic Manager and a Traffic Controller Abstraction have been implemented in order to control and manage the outbound bandwidth of FAIN ANs. These two components are running on user space and their implementation was based on the component model of virtual environment management framework. Traffic Manager derived from the Component Manager and the Traffic Controller Abstraction from the Configurable Component.

For platform we uses the Linux, while the existent of TC in every recent Linux kernel, gives as a very powerful and flexible tool for the building of control mechanism of the bandwidth, which is totally aligned with the requirements of FAIN RCF for the resource controllers. Linux TC has been built on the following major conceptual components:

- queuing disciplines: controls how packets enqueued on a network device (e.g. a network interface) are treated.
- classes (within a queuing discipline): are used in classful queuing disciplines to determines different treatments for different kinds of traffic.
- filters: are used for distinguishing among the different kind of traffic.
- policing: is used in policing filters that only match up to a certain bandwidth.

In FAIN traffic control we make wide use of the three first concepts. For queuing discipline we chose to use the classful and famous CBQ [16]. For every VE we create a different class bounded with specific bandwidth and isolated from the rest classes. Generally CBQ allows the bandwidth borrowing between different classes, but we choose to disable that feature for the VEs' classes. After that VE owner can choose to use between different resource sharing models that are supported by Linux TC. For example, he can create different classes, with specific bandwidth every class, and assign by the use of specific filters, each class to a different flow, or with the use of the TOS field or the DSCP bits the packets can be filtered in a way of mapping different classes to aggregation of flows.

The decision is up to VE owner, but RCF is responsible to check the validity of every request, which is something that is not supported by Linux TC. Also with the cooperation of FAIN Security framework (see chapter 5) every access to Linux TC is checked against the authorization and the privileges of the requested entity.

In Figure 3-5, the involved components that have been implemented and their interactions are depicted.

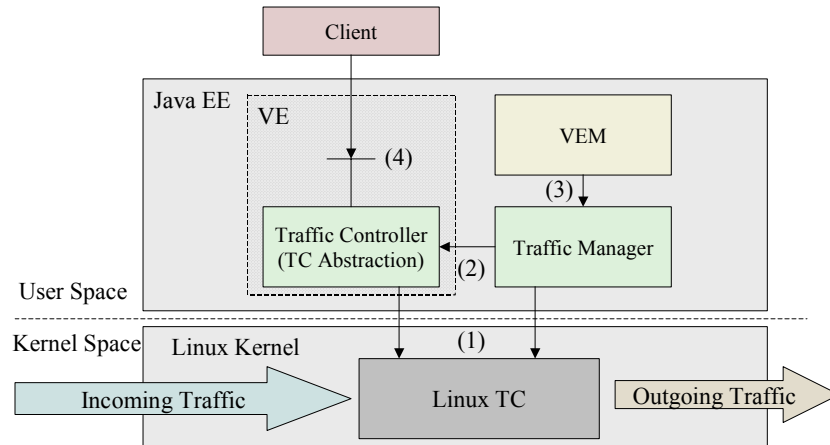


Figure 3-5: Traffic Control and Management for Linux Software Router

As it is shown in Figure 3-5, four are the main interfaces that are used:

- Interface 1: it is command line interface and it is used by Traffic Manager and Traffic Controller for the configuration of Linux TC.
- Interface 2: it is a CORBA communication interface and it is defined by the VE management framework. It is used by Traffic Manager in order to configure the Traffic Controller.
- Interface 3: like interface (2), it is used it is a CORBA communication interface and it is defined by the VE management framework. It is used by VEM in order to request by the Traffic Controller the creation of a new Traffic Controller or the reconfiguration of an existing one.
- Interface 4: Is also a CORBA interface. It is the open configuration interface that the RCF provides to the VE client. It provides a set of operations that offers the capability to the VE client to create finer granularity of the bandwidth allotment by creating new child classes of the VE class and to create classification rules in order to assign traffic to those classes.

The Traffic Manager is the component which is responsible to manage the overall outbound bandwidth of the FAIN AN. First of all it is responsible for the initial configuration of the Linux TC during the bootstrapping of the node. In addition, it divides the bandwidth to the various VEs by creating an isolated and bounded CBQ class for each of them. Moreover, it creates a Traffic Controller for each VE. Finally it decide weather or not the requirements for bandwidth of a new VE can be satisfied or not, and therefore admit or not that new creation.

Every Traffic Controller is the abstraction of the Linux TC to the VE client. It is the entity that manages the corresponding to the VE, TC class inside the Linux TC. In addition with the open control interface (Figure 3-5 – interface 4) that exports, provides part of the Linux TC functionality to the VE client, but only in order to have access and to be able to manage the part of the bandwidth that belongs to the VE. When an authorized client calls an operation of the interface, the Traffic Controller checks the validity of the request, checks also if the status of the configuration of the VE TC class justifies that the request can be satisfied and then translate the request with an appropriate sequence of TC commands. The execution of these commands configures the Linux TC in a way that the request of the client be satisfied.

3.4.2 DiffServ Control and Management for a Gigabit Router

For the sake of the DiffServ scenario with the use of a Gigabit Router like GR2000 or TC-100 of Hitachi, a Diffserv Controller and its corresponding Manager have been implemented. The functionality of the DiffServ Manager is very simple as it just initialises a DiffServ controller for every new VE that will be used as part of a virtual private Diffserv network. The description of the Diffserv

controller for a gigabit router and its functionality follows.

3.4.2.1 Diffserv Controller for the Gigabit Router GR2000

The Diffserv Controller component as shown in Figure 3-6 is in charge of dynamic configuration of the Gigabit Routers, e.g. GR2000, TC-100, based on the traffic conditions. To do this, this component provides two main functions, configuration and monitoring. The Diffserv Controller configures GRs with mapping DSCP values on specific flows. The configuration is done using the BANG API that was the result of the Hitachi and FhG collaboration project. The monitoring function consists of the SNMP traps handler and the interaction with the Monitoring System of the FAIN PBNM. The SNMP traps from GRs are captured through the VEM and analysed in the Diffserv Controller, then the Monitoring System is notified based on filters. Eventually GRs are reconfigured considering to the condition.

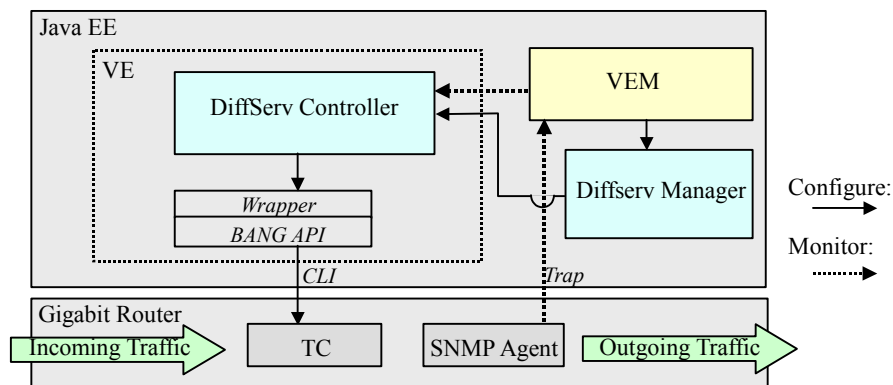


Figure 3-6: Diffserv Controller for Gigabit Router

3.4.2.2 Interface for a Gigabit Router (GR)


We have implemented the interface code (Wrapper) to configure GRs on the top of the BANG API. This interface is to send command by telnet function. `Init()` establishes a connection between Gigabit Routers and an active proxy, then opens a configuration file of the router whereas `closeConnect()` saves and closes the configuration file, then terminates this connection. `BindFlow2DSCP()` is called by DiffServ Controller in order to set DSCP value to the specific flows, e.g. video stream flow.

`setSNMP()` is used to configure basic parameters e.g. snmp community name, that are required by GRs in order to function SNMP on it. `setEvent()` and `setAlarm()` are to set filters for GRs to detect events.

3.5 CONCLUSIONS

In this section the RCF module of the FAIN AN was described. RCF partitions the resources among the VEs. It is responsible for keeping the resource consumption of VEs within the agreed contract. In addition it is responsible to perform Admission Control whether the creation of a new VE can be admitted or not, based on the resource requirements of the new VE and the availability of the FAIN AN resources. The RCF Architecture we introduce is a generic and flexible framework that can support the control and management of various resources of the FAIN AN. We implemented an RCF prototype that can control and manage the outbound bandwidth with the use of Linux TC. For that purpose, two RCF classes was implemented, namely the Traffic Manager and Traffic Controller as part of the VEM framework. Finally, we implemented two special RCF Classes, the DiffServ Manager and DiffServ Controller that are used to dynamically configure the GR2000 gigabit router in order to act as part of a DiffServ network.

4 DEMULTIPLEXING

In active network, an active node receives packet data and processes it. To realize it, packet data should be transmitted to a proper environment for processing in the node. Therefore the active node classifies receiving packet data at first. Then the active node transmits packet data to the proper processing environment based on a categorized class. To classify the packet data, it must have an identifier. For example, packet data might have a specific identifier such as a processing environment ID or classification might be executed based on an IP header data. Someone sends packet data with the environment ID but other one might send packet data without the environment ID. Therefore FAIN active node has to deal not only the packet with the ID but also the packet without the ID. In addition, even  IP data-gram has an environment ID, when the IP data-gram is fragmented, fragmented IP packet data doesn't have the environment ID except a first IP packet data. Therefore active node must handle fragmented packet data. Besides, in FAIN active network, packet data that should be executed processing will be changed dynamically, therefore the active node has to support dynamic updating of policies that include relation between conditions and handling procedures of the data that is classified by the conditions.

The objective of FAIN demultiplexing framework is providing mechanism to realize dynamic updating of demultiplexing policy and processing of packet data regardless of existence of specific ID for processing environment for both receiving packet data and forwarding packet data.

The scope of FAIN demultiplexing framework includes providing an interface for dynamic updating of demultiplexing policies and transmitting packet data to an appropriate processing environment after classifying the data.

4.1 REQUIREMENT FOR DEMULTIPLEXING

4.1.1 Requirement for Active Packet format for Demultiplexing

According to previous surveys, following requirements are listed for packet format.

- Active packet format must include an identifier for distinguishing which data should be dispatched to which VE/EE. For example, we need to decide whether we should send data to VE-1 or privileged VE.
- In addition, active packet format also must include a size of an active packet and a size of an active packet header.

4.1.2 Requirement for Demultiplexing Mechanism

- Demultiplexing mechanism must deal with identifier of VE, EE type of active packet for distinguishing flows.
- Demultiplexing mechanism must send received data to a security component for executing security check before transmitting data to a proper VE/EE.
- Demultiplexing mechanism must create an in channel for sending data to a VE/EE and create an out channel for receiving data from a VE/EE for sending data to the outside node.

4.2 DEMULTIPLEXING FRAMEWORK

The packet data is delivered to a proper VE or service by a demultiplexing function. The packet data includes both ANEP (Active Network Encapsulation Protocol) packet and other data packet (passive packet). The ANEP packet delivers active packet data and the other packet delivers not active data but data for being processed. Figure 4-1 depicts a block diagram of packet data delivery.

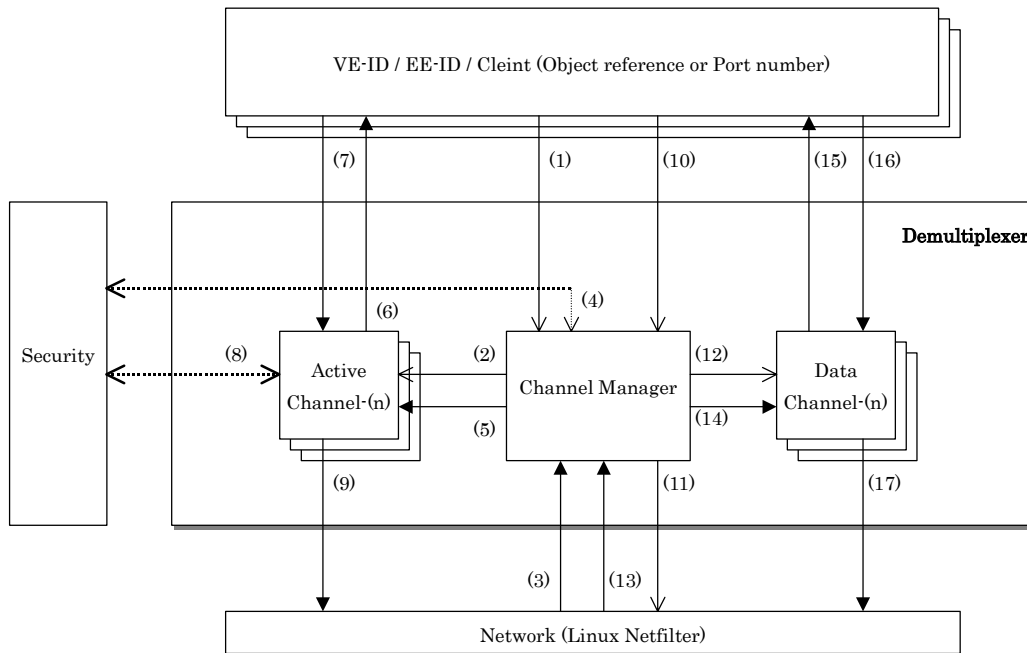


Figure 4-1: Block Diagram of Packet Delivery

Active packet data (ANEP) delivery: (1) At first a client requests a channel manager to create a new active channel for receiving ANEP packet data by registering VE-ID, EE-ID and an object reference of itself or a socket port number for receiving. (2) The channel manager creates an active channel by registering an active consumer object, which includes the VE-ID, EE-ID and the reference or the socket port number, into an internal table for active packets. (3) The Netfilter transmits received ANEP packet data to the channel manager since the channel manager sets conditions to intercept ANEP packets at the booting process. (4) The channel manager calls a security function for checking the ANEP packet before sending it to a proper client. (5, 6) After executing the security check, the channel manager sends the ANEP packet data to a proper client through an appropriate active channel by getting a target to transmit from the table for active packets. (7) If there is ANEP packet data to send to another node, the client sends ANEP packet data to the proper active channel. (8) The active channel inserts the security information into the ANEP packet before sending it to the outside network. (9) After that, the active channel transmits the ANEP packet data to the outside network.

Non active packet data delivery: (10) At first a client requests the channel manager to create a new data channel for receiving data packet by registering flow conditions and object reference of itself or a socket port number for receiving. (11) The channel manager sets the filter conditions, which are given by the flow conditions, to the Netfilter. The filter condition contains which data packet should be sent to the client. (12) Then the channel manager creates a data channel by registering a data consumer object, which includes the flow conditions and the reference or the socket port number, into an internal table for data packets. (13) The Netfilter transmits data packet to the channel manager since the channel manager sets conditions to intercept data packets. (14, 15) The channel manager sends data packet to a proper client through an appropriate active channel by getting a target to transmit from the table for data packets. (16) If there is data packet for sending to another node, the client sends data packet to the proper data channel. (17) The data channel transmits data packet to the outside network.

4.2.1 Active Channel

Figure 4-2 shows how to transmit an active packet (ANEP packet), which is shown in Figure 4-3, to a proper receiver (client). When a new client is instantiated, it needs to register its VE-ID and EE-ID with an object reference or a socket port number. The channel manager stores them into the database for active channel. Table 4-1 shows the contents in the database for active channel. When the channel manager receives the active packet, it checks the VE-ID and EE-ID, which are included as options' data in the active packet as shown Figure 4-4 and Figure 4-5, and searches the database for finding the

target reference or the port number from the VE-ID and EE-ID for retransmitting the active packet. After getting the target, the channel manager checks the target is an object reference or a socket port. If the target is the object reference, the channel manager calls a proper method of the target reference. If the target is the socket port number, the channel manager sends the active packet data to the proper port by the UDP socket.

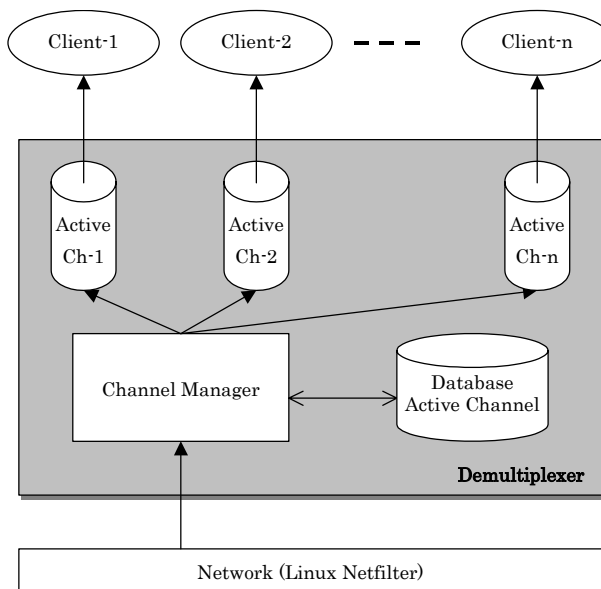


Figure 4-2: Active Packet Transmission

Table 4-1: Database for the Active Packet

No.	VE-ID	EE-ID	Target
1	1	2	Port = 9995
2	1	3	Reference = XYZ
3	2	2	Port = abc
---	---	---	---
n	L	M	Reference or Port

4.2.1.1 Active Packet Format

Figure 4-3 shows the ANEP packet format. We have adopted an ANEP packet format as a FAIN active packet format. We have only added options for the FAIN Type ID. The explanation of each field in the ANEP packet format is as follows;

- **Version**

It means the version of the header format in use. Currently the value of the version is one. This field is 8 bits long.

- **Flags**

In version one, only the most significant bit (MSB) is used. If the MSB of this field is 1, the node should discard the packet. If the MSB of this field is 0, the node tries to forward the packet. This field is composed of 8bits long.

- **Type ID**

It means an evaluation environment of the data. **The value of Type ID for FAIN must be selected.** For demo we suggest to use the number of 10561 as a FAIN TYPE ID.

- **ANEP Header Length**

This data specifies the size of ANEP packet header in 32 bit words. The ANEP header means from the field of Version to the field of Options.

- **ANEP Packet Length**

This data specifies the size of the ANEP packet in 32 bit words

- **Options**

This field is used when there is an option data.

- **Payload Data**

Active code, policy data and data being processed etc. are considered as examples of payload data.

0					31
N	4N+ 0 Byte	4N+ 1Byte	4N+ 2 Byte	4N+ 3 Byte	
0	Version (8bit)	Flags (8bit)	Type ID (16bit)		
1	ANEP Header Length (16bit)		ANEP Packet Length (16bit)		
2--m	Options				
m+1	Payload				

n					

Figure 4-3: ANEP Packet Format

4.2.1.2 VE-ID Option Data

Figure 4-4 shows a format of a virtual environment (VE) identifier. This option is a must when the type ID in the ANEP packet is an identifier for FAIN type ID.

- **FLG**

The owner of EE-ID defines the value of flag (FLG).

- **Option Type**

The value of option type for environment identifier is 101. (For example)

- **Option Length**

The value of option length is 2 in 32 bit words (4 byte).

- **VE ID**

This data means an identifier for sending active packets to proper VE. This field is composed of 8bits. ANSP assigns a VE ID when a SP requests to create a new VE. But the value of zero is reserved for future used and one is assigned for privileged VE.

	0			31
N	4N+ 0 Byte	4N+ 1Byte	4N+ 2 Byte	4N+ 3 Byte
0	FLG	Option Type	Option Length	
1	Virtual Environment (VE)-ID (32bit)			

Figure 4-4: Virtual Environment Identifier

4.2.1.3 EE-ID Option Data

Figure 4-5 shows a format of an execution environment (EE) identifier.

- **FLG**

The owner of EE-ID defines the value of flag (FLG).

- **Option Type**

The value of option type for environment identifier is 102. (For example)

- **Option Length**

The value of option length is 2 in 32 bit words (4 byte).

- **EE ID**

This data means an identifier for sending active packets to proper EE. This field is composed of 32bit. Each VE owner assigns the EE ID.

	0			31
N	4N+ 0 Byte	4N+ 1Byte	4N+ 2 Byte	4N+ 3 Byte
0	FLG	Option Type	Option Length	
1	Execution Environment (EE)-ID (32bit)			

Figure 4-5: Execution Environment Identifier

4.2.2 Data Channel

Figure 4-6 shows how to transmit a data packet to a proper receiver (client). When a new client is instantiated, it needs to register flow conditions with an object reference or a socket port number. The channel manager stores them into the database for data channel. Table 4-2 shows the contents in the database for data channel. When the channel manager receives the data packet, it gets the flow conditions, which means a source IP address, a destination IP address, a protocol number, a source port number and a destination port number, by checking the header of the data packet and searches the database for finding the target reference or the port number from the flow conditions for retransmitting the data packet. After getting the target, the channel manager checks the target is an object reference or a socket port. If the target is the object reference, the channel manager calls a proper method of the target reference. If the target is the socket port number, the channel manager sends the data packet to the proper port by the UDP socket.

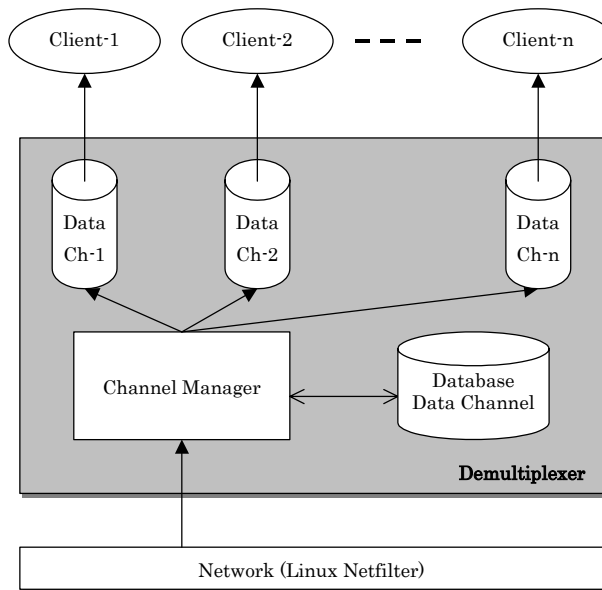


Figure 4-6: Data Packet Transmission

Table 4-2: Database for the Data Packet

No.	Source_IP	Dest_IP	Protocol	Source_Port	Dest_Port	Data Ch
1	sip-1	dip-1	p-1	sp-1	dp-1	Reference
2	sip-1	dip-1	p-1	sp-1	dp-2	Port
---	---	---	---	---	---	---
m	i	j	k	l	m	Ref./Port

4.3 CONCLUSION ON DEMULTIPLEXING

In this section, the architecture of demultiplexing (DeMUX) is described. DeMUX intercepts packet data and dispatches it to an appropriate receiver client. It provides a virtual channel for each VE for packet data transmission between DeMUX and receiver client. To realise the function, Channel-Manager and Channel classes are implemented. The Channel-Manager creates one virtual channel object for each VE. The virtual channel includes multiple active channels and data channels. The active channel is used for active packet data transmission and the data channel is used for non-active packet data transmission. The definition of active packet data is whether it includes data for node configuration or programs, or not. When a receiver client receives packet data from DeMUX, it needs to register a target object that includes receiving method and flow definitions. Currently DeMUX supports CORBA interface and simple socket interface as packet data transmission method. Concerning about flow definitions, especially active packet data, the client can register VE identifier and EE identifier that should be included in the ANEP packet as options' data. In addition, in case of non-active packet data, the client can register five tuples (Source IP address, Destination IP address, Protocol number, Source port number and Destination port number) for classifying a flow. DeMUX retransmits an intercepted packet data to the proper receiver client according to previous flow categorization. In addition, it calls security check for active packet before transmitting and if the security check falls in failure, it discards the active packet. Therefore DeMUX can provide flexible and secure demultiplexing function to the receiver client.

5 SECURITY

5.1 INTRODUCTION

Programmable and active networks enable their users to extend and program network elements to fulfil their specific communication needs. FAIN aims to develop a flexible, high performance and secure active network node. FAIN architecture allows various active networking technologies to be used in the same node with aim to be able to implement various services for its users in transport, control and management plane.

Flexibility of such system raises serious security concerns. Mainly in active networking, security is an area of intensive research already for more than half a decade. The security solutions in general can be divided in two distinct approaches: architectural based and language based. Architectural based aim to provide more or less complete security solutions like U. of Pennsylvania SANE [19][20] or Active networking security working group security architecture [5][22]. Language based is relying on safe language and interpreter design and achieve security through seriously limiting the ability of the programs that can be injected in the network [31][33]. FAIN as approach using multiple technologies can benefit from language based approaches but cannot rely only on them; general security architecture has to be provided.

FAIN aims to develop a complex environment for flexible service provisioning and management of these services; existing security approaches are targeted to more simple structured environments, do not cover collaboration of multiple EEs, services and service components, neglect management issues, do not provide clear view of system entities or cover only a part of the tasks that the security architecture should perform.

Security architecture is a set of principles, services and mechanisms that are required to meet the needs of its users, prevents intentional and unintentional threats and set of system elements that implement the services. To define the needed principles, services and mechanisms we will introduce system relationships and entities in section 5.2, threats, security relationships and architecture goals in section 5.3, discuss security issues in section 5.4, propose high level security architecture and introduce system elements in section 5.5. Security architecture design and implementation will be presented in section 5.6, general scenario of an active packet passing a node presented in section 5.7 with report on system performance in section 5.8. Whole approach is evaluated in section 5.10 and architectural applicability in the case of existing active networking approach is presented in section 5.9. Finally we conclude with our conclusions and directions of future work in section 5.11.

5.2 SYSTEM RELATIONSHIPS AND ENTITIES

The basis of the node is defined with FAIN reference model [2][22], decomposing node in four layers: router or hardware, node operating system (NodeOS), Virtual Environments (VE) and services. NodeOS is a collection of basic node services, which performs tasks of demultiplexing, resource control, active service provisioning, security and management. NodeOS functionality is exported through NodeOS API to Execution Environments (EE), which resides in VEs. VEs are resource and user related abstraction on the node; in a VE reside one or more EEs and services. Services are collection of components performing an application for its users. Service and service components are defined by a service descriptor which is resolved in the network and on the node in one or more code modules which can be instantiated into certain EE(s) where the component(s) become a runtime instance(s).

For example, FAIN node supports Java based Execution Environment (EE), High performance active network node [26] and SNAP [33] as an EE. These environments can be used in synergistic way to support or complement each other. High performance environment can be supported by Java based EE for management and control of the former and SNAP is extended with the Java environment to manage SNMP enabled devices.

For described model, on the network element, packets are interpreted as requests and evaluated on the node, within a service in one or more components, and can result in zero, one or more packets on the

output of the node. Requests or result of an evaluation can be either active or passive packets. We consider as active packets those packets that contain ANEP header as defined in ANEP draft standard [1]. Packet content or the state on the node can be changed during evaluation. Code or data in the packet can result in action regarding the API defined by dedicated component Execution Environment or NodeOS API.

Service can exist in a single node or can span over multiple nodes in the network. Packets exchanged between communicating entities, can be processed only at the sender and receiver, or, depending on the nature of the service, on any suitable node in between. For purpose of our discussion we will assume a simple network structure as shown in the bottom of the figure 5-1. Structure itself is virtual; there can be also passive nodes in the network as well.

Code modules extending programming environment of the node are result of service deployment in the network and service resolution on the node. Such approach is called out-of-band because the code is not transferred together with data; to gain additional flexibility FAIN uses also in-band approach in Active SNMP service, reusing existing active networking approach, the SNAP [33]. But in FAIN case majority of the code can be considered out-of-band.

From the system relationships we can deduce the following set of entities in the system: network users, network nodes and the services. Execution environment in the processing model is a technical term defining needed system elements for successful evaluation and it is not real system entity per se. To be able to abstract the resources that the active network user has available on the node and its services are using, FAIN use Virtual Environment abstraction (VE). VE represents resources on the node and environment needed for the service operation. VE is a flexible abstraction; it can overlap tightly with a service, a specific user or it can be used in the context of multiple services and users.

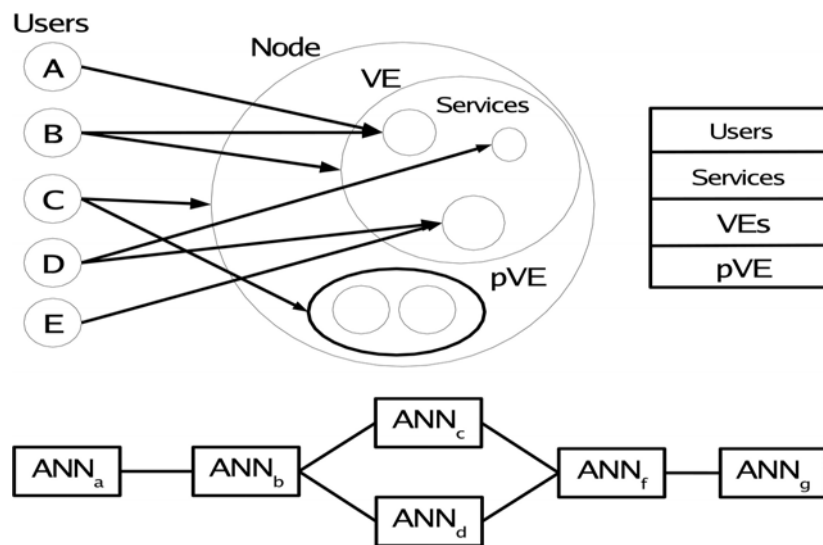


Figure 5-1: System entities and relationships

The relationships between entities are shown in figure 5-1. The figure shows two types of relationship: one is the ownership showed with arrows and the other is the relationship between system entities on the node regarding the resources shown with circles. The set of users on the left 'owns' resources on the node. User C is an owner of the node and thus owner also of the privilege VE (pVE). pVE is a special VE abstraction which holds basic node services, besides others also security related, as will be explained later in section 5.5. User B is an example owner of a VE, which can run one, or more services associated with a VE. Users D and E 'owe' same service; actually this case, and also other related to services, it represents not direct service 'ownership' but the ownership of the packet evaluating in the context of the service. Of course there can be as many VEs on the node as node can handle in the sense of available resources.

On the right side of figure 5-1 system entities are shown; pVE corresponds to node owner, VE either to service provider or group of related services, services can be either user or application related and

finally users of these services, which can be service, VE or node managers, users, observers etc. These entities can be easily related to those defined in FAIN business model.

In the system there can be also a number of 'external' entities that support the system operation. Such entities are for example code producers, certification and attribute authorities etc.

5.3 THREATS, SECURITY REQUIREMENTS AND ARCHITECTURE GOALS

Threats in described system are considerably extended regarding traditional 'passive' systems, where evaluation is more or less related to forwarding a packet through the network element. But threat consequences are no different as in traditional systems and are, as categorized in [38]: disclosure, deception, disruption and usurpation. Reasonable selection of threat actions that can cause a threat consequence applicable to a described system can be no other as presented in [21].

Entities in the system that can be a source of the threat action causing a series of threat consequences are all entities presented in the section 5.2. pVE represents a threat to all other entities in the system. While basic pVE services (security, resource control) can prevent threat actions between other entities the threat represented by pVE can be prevented with strict control which nodes can process and evaluate packages.

Basic security related problems can be seen in the following areas: startup of the node and connecting it to the network, partitioning of resources of the node and in the network, deployment of new services, related code, configuration and policies in the network and on the node, accessing, managing, control and observing the services and its related data, control which data (packets) a service can access, naming of the resources in the network, providing of traceability of the node state and protection of the basic node resources.

Security architecture designed in FAIN has the following general goals and requirements:

- *authorized use*, protect network element and user resources in the network. Network element resources can be functional, computational or communicational. User resources are the packet content and the possible states on the network nodes. Only authorized users should be able to access these resources, and only authorized nodes can process the packets,
- *separation* between different VEs and their related services regarding access and resource usage should be enforced in the network and on the network elements,
- *verification*, code brought to the node has to be verified either statically or dynamically and has to be protected on the node together with code related configuration and data against intentional or unintentional changes,
- *accountability* of security related events, audit service should be provided on the nodes and in the network,
- *protect in transit*, security architecture perimeter is a network element; communication between network elements should be protected and provide in secure manner sufficient information about communicating entities in the system for security architecture operation,
- *common treatment* regarding security of: (a) network elements, like end, intermediate, management nodes, (b) VEs, services, components, (c) EEs and code modules, (d) and communication in between and across multiple network elements, (e) same security mechanisms should be used in the management, control and transport plane,
- *transparency*, operation of security architecture should be transparent to its users and developers, either through well defined interfaces, protocol headers or architecture implicit operation and should require minimal user intervention,

- *flexibility*, (a) multiple trust management approaches should be supported between entities in the system, (b) multiple types of security policies should be supported, (c) security architecture proposed should be general enough that it can be used for all developed technologies in FAIN and also existing established technologies,
- *sufficient and extensible*, basic security services should be sufficient for safe network element operation; but it should be possible to extend the security architecture by certain VEs or services to fulfil their specific needs.

Security architecture goals can be achieved mainly with authentication, authorization and policy enforcement, system, code and packet integrity service, code verification, limiting users resource usage on the node and in the network, audit service, right choice of selected security mechanisms and system design. These issues are discussed in the next section.

5.4 SECURITY ISSUES

5.4.1 Authorization and policy enforcement

All accesses to the node and user resources should be subject to authorization. Node and user resources can be hardware like CPU, memory, storage and link bandwidth or functional like special purpose files, routing tables, policy and credentials entries and databases, VEs and service related data etc. Important resources are possible service states on the nodes, which can be shared among multiple users, and user packet content.

Authorization is a process that provides an authorization decision about access of the subject to the object. Decision is provided to enforcement engine. In general we assume that when the subject accesses the object enforcement engine suspends the request and asks authorization engine for authorization decision. Information passed to authorization decision is security context of the subject and object, action and possible environment of the access. Based on the passed information the authorization engine returns authorization decision that is then enforced by the enforcement engine. Security context is all security relevant data available on the system regarding subject and object. Of course proper authentication is needed to provide authorization.

Policies control which users can have access to certain resource and in what manner. Policies should be detached from the system and should not be hardwired in the application. Because the used policies in general can be various, the system should be flexible enough to support multiple kinds of policies.

Users in the context of policies can be defined either by their identity or attributes like roles or groups. We will name both user attributes; users in the system should possess certain credentials, which express these attributes. Multiple type of credentials should be possible to enable various ways of trust management between users. Scalable approach should be provided in the system to enable nodes to access user credentials. When transferring through the network credentials integrity must be provided. If credentials are referenced or included in the packet such information must be bound to the packet content. Replay protection for such packets should be provided.

It should be possible to generate and enforce certain policies by the system itself, independently of the fine grain user policies. Such policies can be understood as mandatory access policies that enable separation between system entities like VEs and services and prevent their unneeded interaction.

Certain kind of policies can require that the state of previous authorization decisions is known. System should be able to define such state and to keep it available for later authorization decision provisioning.

It is hard to believe that all policies and all users can be globally understandable. But we can assume that the users and the policy can be known in single administrative domain or in the context of a certain service. System should be able to provide functionality that can enable replacing of the user credentials on the administrative domain borders.

Enforcement process must have classical attributes of a reference monitor: it has to be non-bypassable, tamperproof and analysable [27]. Besides, as is correctly noted in [20], it has to be non-spoofable.

Security context of the mentioned subject and object has to be bound to them in such way that cannot be forged in the system. We can add to this that authorization decisions made, based on the security context, are valid only as much as is valid the data in the context. So the process of building a security context has to be carefully examined to be able to provide authorization decisions based only on the data we can trust.

5.4.2 Authentication

Authentication is a needed service for authorization and also for other security services in the system. Authentication has to be provided per packet, connectionless, because of the assumed model of communication.

The major problem of authentication service in active networking approaches is how to authenticate an active packet passing many nodes. For authentication symmetric or asymmetric cryptography based solutions can be used. Using symmetric cryptography solutions requires that the session key is negotiated or provided to two or more parties in communication. Asymmetric based approaches do not require such step but the trust has to be managed between the active packet sender and the nodes that the packet traverses. Symmetric approaches do not provide non-repudiation; this fact is important if two or more nodes use the same session key and any of them can become a source of the authenticated packet. In such setup, if one node is compromised, this can be a serious security problem. Besides, session key can be seen as a hard state on the node; if it is not available, communication will fail. Symmetric techniques that negotiate separate session keys with each node in its path and provide authentication data for every node in the packet are too costly in the terms of bandwidth and negotiation time. Approach has same problems with packet integrity as using asymmetric techniques as described in section 5.4.3.

Symmetric cryptography approaches are still very useful when used properly; neighbour nodes can identify each other in this way after they have established trusted relationship and negotiated security association (SA). Dependent on protection mechanisms used in SA they can provide besides authentication of a peer also integrity and confidentiality for the packets exchanged. Such types of communication we will call a session; they are useful for inter node communication or for communication of a user with a node. Concept of sessions can be supported in various way, it can be based for example on IPSec or SSL.

Data origin authentication is related to the data integrity; there can be no data origin authentication if data integrity is not provided as well.

5.4.3 Packet integrity

Packet content can be legally changed in the network. In this context we will concentrate on active packets; while also passive packets can be changed in certain services, care has to be taken not to interfere with end-to-end security solutions like IPSec.

Changing the packet content raises the question of data origin; if the packet is originating on one node but the content is added to the packet (for example, data is collected on nodes) or something is removed (for example some lines of code) this data as a whole cannot be authenticated for a data origin on the nodes that the packet passes any more. Logical consequence of the stated is to split the packet in the part that can change, e.g. the part that is variable, and in the part that is static during the packet lifetime in the network.

The variable part of the packet can be used by the service to store or modify the packet content. But we still want to protect somehow this variable part from the unauthorized or malicious modifications. First countermeasure is to control which nodes can process and evaluate such packet. With this approach we can avoid unauthorized modification in between two authorized nodes. We will call such protection per hop protection in the contrast to the protection, that can be applied end-to-end for static parts of the packet. Per hop protection is similar to the sessions, introduced in section 5.4.2 and can be understood as system level protection of active packets exchanged between two nodes. Second is, how to control additions or modifications? Because of the diversity of approaches and internal service

knowledge of the data structure it is probably needed to shift this responsibility from the system layer to the service layer.

If the active packet contains a code and data intermixed the integrity of such packets is hard to provide and it is hard to decide which parts of the packet data is static and which variable. Protecting such packets in between hops is helpful but not enough. We will propose later in the section 5.9 a solution for an Active SNMP system.

5.4.4 System integrity

System integrity service is one of the core services that must be provided on network element. It has to be provided from ground up, from the first piece of code run on the node. System integrity guarantees, together with other security services, notably authorization, policy enforcement and code and service verification, that the node will perform its intended function in an unimpaired manner.

The task of the system integrity is threefold: first, when the node programming environment is extended, the state of the code after authorization and possible verification is stored safely, second, during the node operation it enables mechanisms that enable preventing malicious, unintentional and unauthorized system changes, and third, it has to keep previous states of the code and related data so the extensions and modifications of the node environment can be traced or rolled back if required.

5.4.5 Code and service verification

Code and service verification is a security service, which verifies the correctness of a service and code modules operation. We can divide the verification in two broad groups: dynamic and static. Static verification is done prior the injection of the service in the network. Dynamic verification happens prior or during the evaluation; it has to be extremely fast in contrast to the static verification.

As we said in section 5.2, majority of the code comes in a node out-of-band. For out-of-band approaches it is possible to verify the code in the static way. Many kinds of verification are possible like source code inspection, code testing, generating proofs [34], which can be verified on the node prior to code installation and usage etc. For static verification it is important that the trust between the verifier and the code user (node) is established.

Dynamic verification is usually performed by interpreters like in case of Java bytecode verification. When the interpreter model is extended the verification has to be extended also. Dynamic verification is important in the case of in-band deployment of the code.

Specific to service verification in FAIN is how the services are composed; many code modules can be composed into single service when the service descriptor is resolved on the particular active network node. While service can be verified for the desired properties at network level, final verification of the service has to be done at the node level. For every piece of code verification is needed; a common approach should be chosen because of the various active networking technologies FAIN is using.

5.4.6 Limiting resource usage

Resource usage is a hard problem because active service resource usage cannot be clearly determined in advance; in today networks basic IP packet forwarding is proportional to the packet length but every processing that has to be shifted from the hardware path to the network element processor is painful for the network elements. This is true even more in active networking and undetermined resource usage applies both to in-band and out-of-band approaches; neither can guarantee strict resource bounds in all cases regarding the packet(s) that triggers the service.

Basic approach we are planning is limited resource usage per active network user; user resource box as a collection of the available communicational or computational resources per user or class of service has to be known in advance on the nodes in the network. If the limited resource usage can be applied per user on the active node also the network resource usage cannot exceed the user resource limit. There are two important issues in this approach: users' resource usage has to be tracked in all drains of resources per user and NodeOS must have dedicated resources available in advance to function

properly even in cases when all node resources are exhausted.

Regarding the active service, which is triggered by matching packets, the resource usage has to be limited network wide; this can be achieved by tracking the packet integral resource usage count, or how many nodes the packet has passed, and its proportional usage, how many packets the single request produce on the node. In this case the packet available resource limit has to be divided in between child packets.

5.4.7 Accountability

Accountability is important property of the system; it enables us to track and analyse possible security breaches through audit service. Accountability should not be provided only on the node; active services span many nodes and can be influenced by many external subsystems and should be in a single administrative domain gathered in such a way that analysing can be possible from central point.

5.5 HIGH LEVEL SECURITY ARCHITECTURE

From security requirements discussion we can propose a high level security architecture as shown in the figure 5-2. Architecture places requirements as presented in section 5.3 in the system discussed in section 5.2.

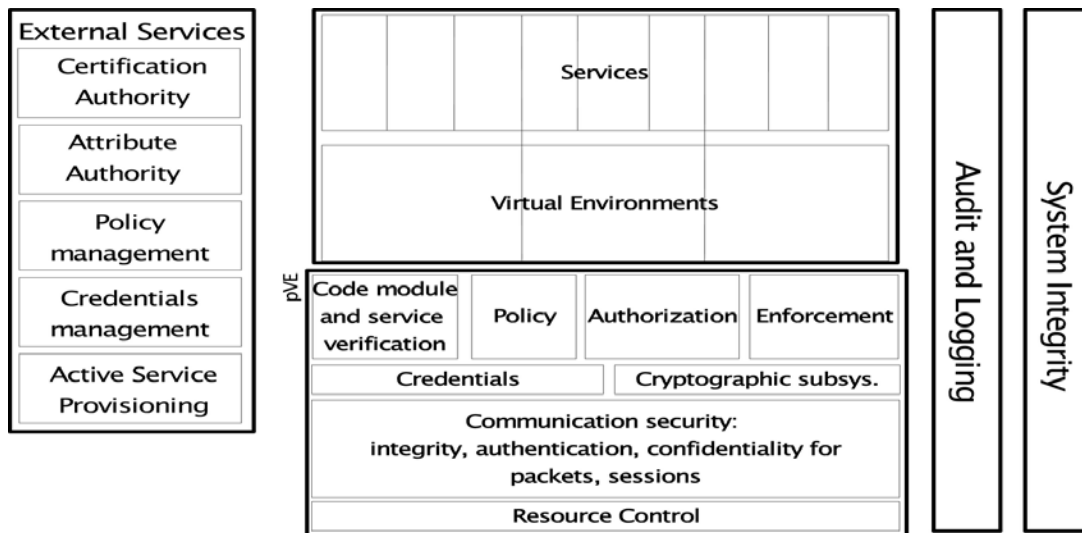


Figure 5-2: High level security architecture

Basic security services are positioned in the privilege VE because of the following reasons: we want to treat all possible technologies and their implementations, implementing VE and services in the one and only one manner, reducing the risk of multiple implementations, and the services offered in the pVE are protected again with the same services and mechanisms. This doesn't preclude VEs or services from implementing their own security services or mechanisms when it is reasonable to do so.

Resource control is not really part of the security architecture. It is a needed element of any reasonable network node; using such element, security architecture can efficiently enforce separation between VEs and services regarding the resource usage.

5.6 FAIN ARCHITECTURAL MODEL AND SECURITY ARCHITECTURE

The core of the FAIN architecture is active network node. Basic functions of the NodeOS were decomposed to the following subsystems: demultiplexing (DEMUX), resource control framework (RCF), active service provisioning (ASP), management and security. Each subsystem is also related to security:

- DEMUX subsystem is responsible for management of input and output channels to VEs and services. At this point the security perimeter of the architecture starts and the security context of the incoming packets is built. At the output channels the packets leave the perimeter and here

is the point where external security representation has to be added to the packet. On the other hand DEMUX export interfaces to set up or tear down the channels; these interfaces are part of NodeOS API,

- RCF subsystem is responsible for resource allocation and enforcement of the resource usage. It enables separation of system entities regarding the communication and computation resources. Guaranteed share of the resources has to be provided to pVE so the basic services can operate uninterrupted. RFC exported interfaces enables resource reservation and report resource usage,
- ASP subsystem is responsible for deploying the code for the service operation to the node; it has to cooperate with the security subsystem to ensure system integrity and static service code modules verification,
- management subsystem is responsible for management of the basic node services, VEs its services and service components. It exports interfaces for their initialization, setup, control, suspension, observation and termination.

High-level security architecture was decomposed to system elements performing needed architecture tasks and mechanisms:

- principal manager, which is responsible for principal related operations: adding to, removing from and searching for principal in principal database. Principal entries are collection of principal related data; principal attributes, list of his credentials and pointer to principal secure store,
- credential manger that manages principal related credentials in an uniform way irrespective of credential type. It is responsible for parsing and validating credentials and extracting principal related credential information; user attributes, credential time validity or possible policies embodied in the credential. Credential manager provides also utilities for user credentials and keystores generation,
- policy manager that manages in uniform way various policies on the node. Policy manager provides also policy engine(s) that can provide authorization decision related to the particular policy,
- security manager which is a central point of the subsystem. It is responsible for building of the security context of the subjects and objects on the node; security contexts are kept in the security subsystem and never leave the subsystem,
- authorization engine that provides authorization decision to node enforcement engines. Authorization decision is based on one or more policy engine decisions. Authorization engine also keep state of the authorization which can be exported to audit subsystem or stored if the certain policies requires such state,
- enforcement layer, which enforces authorization engine decision. Enforcement layer is separated into enforcement engines, where authorization decision is enforced and the mechanism, which enables secure gathering of the subject and object information on the node,
- audit subsystem, which provides audit service on the node through audit channels to audit database.
- system integrity subsystem that collects all code modules, service and EEs related data and reacts to code or service changes, code time validity or code related policy changes,
- verification manager, which enables dynamic verification of the code or services,
- cryptographic subsystem, which offers needed mechanisms for all cryptographic operations,
- principal secure store which holds principal related cryptographic information and provides asymmetric cryptography mechanisms together with cryptographic subsystem in a manner that the principal related private keys never leave the store,

- external security representation subsystem that can build and extract principal (packet e.g. request) related security context information in uniform manner and mechanisms to fetch the principal related credentials on the node,
- connection manager, that can build in secure and trusted way the security associations with its neighbouring nodes, to provide hop protection between nodes,
- integrity subsystem, that ensures packet integrity between two neighbour nodes.

5.7 SECURITY ARCHITECTURE DESIGN AND IMPLEMENTATION

FAIN active node supports various technologies; operating system used is Linux, the core of the node is coded in Java and CORBA is used for communication between node subsystems through set of well defined interfaces in IDL. Subsystems like SNAP, High Performance Network Node and Linux subsystems like Netfilter and OS resource control features were wrapped with Java and CORBA to be able to access, control and manage these environments.

From security perspective is most important that for main system abstraction component-oriented model was used. Everything on the node, including base pVE services, VEs, EEs, services and their components and the active packets are treated equally through this model. Modularity, same treatment of all system components and fine granularity of the approach are beneficial to the security.

5.7.1 Building components security context

For each component in the system during component initialisation and startup security context of the component is built. As shown in the figure 5-3, the process of service or VE startup is started with transition and labelling. Relevant pointers to VE, service and parent VE data are attached and if specific component policy is specified, this policy is set. The security context data is stored in security context database in the node security area, each component gets during initialisation a node local unique opaque id which points to the context.

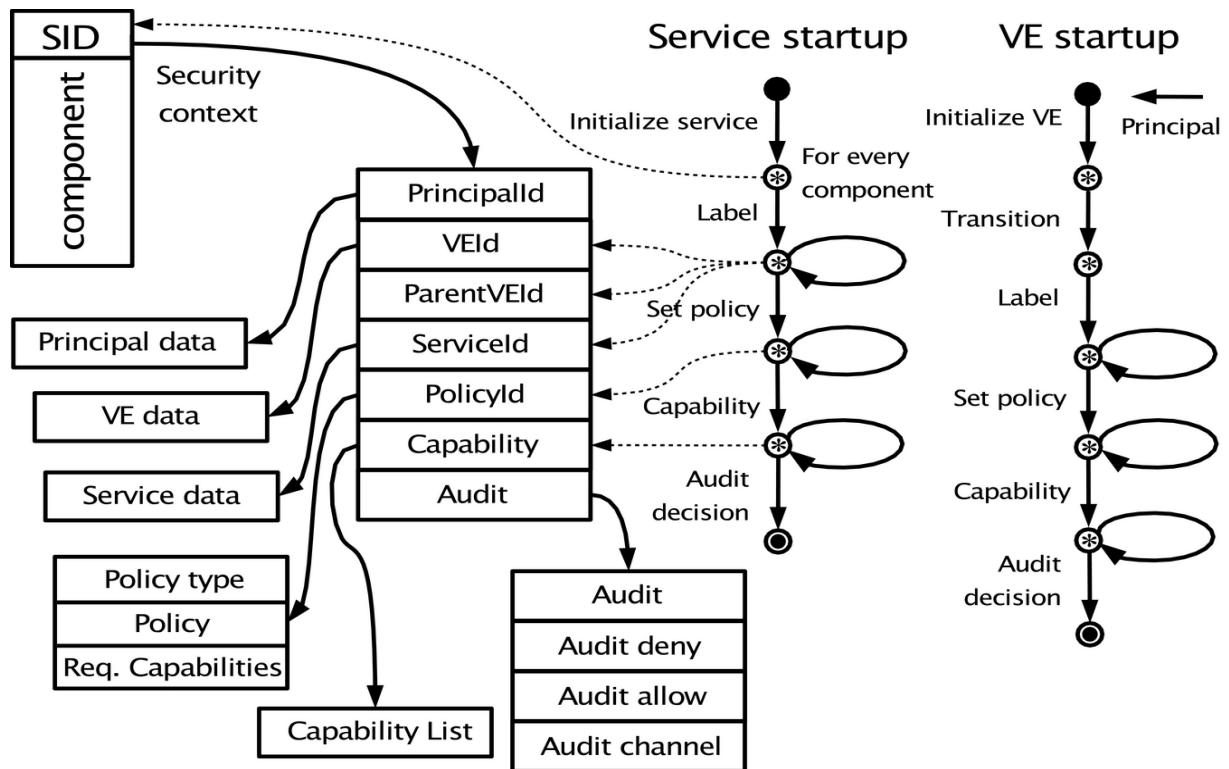


Figure 5-3: Security context, VE and service startup

5.7.2 Enforcement layer, authorization and policy enforcement

When the components communicate the CORBA interceptor is used to transparently pass the security identifier (SID) from the subject to object. When a component interface or its certain ability is accessed they are protected by authorize call which is passed to component authorize method which calls the security manager interface for authorization with SIDs of the communicating components, action and possible environment of the call as parameters.

Authorization engine compares first the VE and service identifiers and if they are identical, the action and possible environment of the call is evaluated in the policy engine, specified by policy type. Authorization decision is returned to the caller.

Active packet is treated in the context of the node in exactly the same way as a component. When his external representation, described in section 5.7.3, is accessible, security context of a packet is built. Packet security context is used as in case of any component to provide authorization and policy enforcement regarding packet related request to system resources. In the same manner access to the packet or parts of the packet can be controlled on the node.

Security again benefits from component-oriented model and object oriented environment. When policies are written for basic components and the authorizations are in place, all inherited components have to take care only about their extensions to the model, which are component specific. The developer of the component has to take care that its abilities can be protected; the task of the management environment is to prepare proper policies regarding their security model.

Sessions and per hop protection in between nodes are treated in similar way as in case of components regarding the security context. Their security context is created from connecting entity credentials and assigned to proxies on the node. In the case of per hop protection the security context is assigned to specific channels related to the node intercommunication services. Example of such service is connection manager case explained in section 5.7.5. Otherwise we also support sessions to the node, which are used in this case to support management stations and their connections to the node. For such connections we use SSL and CORBA; one or more connections are possible with different security context between management station(s) and the node. Security context of such sessions is attached to the component ports returned to the client acting as a proxy for connected user. Context itself is build from user supplied credentials during SSL session negotiation (X.509 certificates).

5.7.3 External security representation

As a basis for external security representation we use the ANEP [1] encapsulation protocol. Solution is applicable to any active networking approach that can be encapsulated with ANEP protocol. Regarding the discussion in section 5.4 we have defined six new options to carry security related information over untrusted connections. These options carry VE and service identifier, hop protection and credentials option related information, service variable data and resource usage information. From original ANEP options in [1], only source and destination addresses are used.

Hop protection is defined by Security Association (SA) identifier, which points to right association with a neighbour node, sequence field that protects against replays and keyed hash. Keyed hash covers the entire ANEP packet except the keyed hash itself. The hop protection protects all active packets exchanged between two neighbour active nodes. As a system layer protection, these fields are removed from the packet after successful check; only information about previous hop node is kept for the packet. If the packet leaves the node new hop option is build regarding the next hop SA.

Credential option is defined by credential identifier and type, location field, specifying where the credentials can be fetched, target field where the user can specify specific targets as nodes, system layer or a packet itself, optional time stamp which protects against replays and the digital signature. The digital signature covers only static data of an active packet: first 32 bits of ANEP header, protecting typeId of active network nodes, source address, VE and service Id, ANEP payload and credential option itself except digital signature data. Time stamp in the credential option is additional measure of protection against misbehaved or subverted node services. Per hop replay protection in this case is not sufficient. For such a service is easy to store and replay the packet later. Roughly

synchronized node clocks are needed for such protection in the network. There can be zero, one or more credential options in a single active packet. On each passing node credential option related credentials are fetched, certification path validated and digital signature in the option verified. Digital signature mechanism enables authentication of data origin, provides data integrity service for the covered data end-to-end and enables non-repudiation. From credential option(s), if present in the packet, a security context(s) is built on each passing node, which is later used for authorization and policy enforcement. Credential types can be various, from X.509 certificates or attribute certificates [18], SPKI [30] certificates or Keynote credentials [24].

Credentials can be fetched in multiple ways, if not included directly in the packet: either from DNS [28], LDAP [23] or any other suitable store. In our case we have designed and implemented simple protocol that enables fetching credentials from the previous hop node. In this way, it is sending entity responsibility to supply all needed credentials that can be validated later on the nodes that packet traverse. To be able to supply credentials on the intermediate nodes we have designed and implemented node credentials cache. After successful validation, the credentials are cached on the node for the time of their validity or regarding the cache policy about cache size and maximum time period of the cache entry. Caching credential has also other benefits; if the cache entry is valid, there is no need to validate the credentials. In this way we can reduce required digital signature validation to only one per credential option in the packet, which result in significant speed improvement, after the first principal packet has passed the node. Additionally we cache also bad credentials in the separate cache in the cases when the credentials cannot be verified. Packets with such credentials are discarded immediately. The same mechanism could be used with a supporting protocol embodied in service for exchange of bad credentials and preventing temporally or permanently access to the node for certain principals with very low cost per packet.

VE and service identifiers are used by demultiplexer to divert the packets to the right service. Variable option is used by the service to store the data that can change in the network, ether its state, collected data etc. Resource related option is at the moment only a simple counter of the passed nodes.

5.7.4 Cryptographic subsystem and secure store

For cryptographic subsystem we have used Java based and Sun JCE compliant cryptographic library.¹ The part of cryptographic operations is performed inside secure store that wraps the digital signature related operations and Java keystore functionality in a way that users' private keys never leave the store. Only pVE security subsystem components have access to the stores; user stores can be managed directly by the users but are additionally protected with the password.

5.7.5 Connection manager

Connection manager is responsible for setting up Secure Associations (SA) between neighbour nodes.² It exports interfaces so that the SAs can be managed either manually or by triggering key exchange by the Network Management System. Additionally, protocol was designed to exchange the keys automatically. Protocol reuses the same mechanisms and possible credentials as discussed in section 5.7.3 and Station to Station protocol as described in [36]. Protocol is modified to this extent that entire protocol messages are covered by digital signature in credential option. Messages are addressed to the channel that doesn't provide hop protection but access to it is authorized. Two nodes, as protocol entities establishing a SA, has to supply credentials, that contain suitable authorization information regarding the channel policy, to succeed.

5.7.6 Verification manager

For static verification digital signature mechanism was chosen as the most common one. Static verification is used in the process of out-of-band code deployment to the node in conjunction with

¹Bouncycastle Java Crypto library, <http://www.bouncycastle.org>.

²In the sense of the network topology in figure 5.1, which represents a virtual topology, such topology has to be build and managed dynamically. Such work is not covered by security architecture.

Node level ASP manager. The cryptographic hash of the code is digitally signed and code digital certificate issued by either code producer, verifier, or trusted archive. The verification manager verifies the credentials certification path and the signature; the authorization decision about possible code deployment is made with common developed authorization mechanism regarding the local node code repository policy.

Dynamic verification can be added as part of Verification manager functionality as described in section 5.10 for Active SNMP system.

5.8 GENERAL ACTIVE PACKET SECURITY EVENTS

In general case we can divide active packet security events in three parts: entry-level checks, evaluation level checks and exit level checks.

Entry level checks steps are the following: after an active packet is diverted by demultiplexing subsystem, if it recognized as active packet, the packet is passed to security subsystem. Based on SA identifier in the hop protection option, right SA is selected and sequence replay protection is checked. Option keyed hash is verified and if verification is successful the resource option is checked for maximum number of hops. After that credential option is parsed, credentials are fetched from the previous node if they are not in the credentials cache already. If credentials are successfully validated they are stored in the cache and the digital signature in the option is verified. Credential timestamp if present is compared to the local clock. Security context is built from this credential option. The procedure is repeated for every credential option in the packet. VE and service identifiers in the packet are compared to those stated in the credentials. If they match, packet as a request is authorized against the security context of the input channel. At least one security context must be authorized positively for the packet, otherwise packet is dropped. If the incoming channel requires code verification the code is verified. If the packet passes all checks it is returned to the demultiplexer, which sends it to the service.

Evaluation level checks are performed if the packet evaluation results in access to NodeOS interfaces, service state or packet itself. In these cases actions are authorized and service or node policy enforced as described in the section 5.7.2.

Exit level checks. Basically, when the packet is sent by the service to exit channel demultiplexing subsystem invokes security send check interface and the packet resource counter is increased, right SA regarding the packet next hop destination is selected and the hop option is built and inserted in the packet. The packet is returned to the demultiplexing and sent to the wire.

In the context of the exit level checks there also other checks possible. The exit channel can have a policy set that can be evaluated regarding the packet security context(s). Resource counter is divided among outgoing packets when the evaluation on the node results in multiple packets.

5.9 SECURITY ARCHITECTURE PERFORMANCE

We are mostly interested in security architecture performance in the case when the active packet passes the node. For this purpose we assume that the suitable VEs and services are already setup on the node and that the user can access secure store and his credentials and related key pair to be able to create ANEP packets and corresponding credential options in the packet. Originating node should have already established SA with his neighbour node. On the nodes that the packet passes security costs are related to the general scenario as described in section 5.8.

Testing environment was setup on commodity PC, with Intel P4 2.2 GHz processor with 512 MBit RAM, Red Hat Linux 8.0, kernel 2.4.18-14, Java SDK 1.3.1_3, Bouncy Castle crypto-library version 1.17 and network node related FAIN code.

Figure 5-4 shows the security related cost of an active packet passing the active node. Active packet in used case contains basic ANEP header [1], hop option, option for VE and service identifier, one full credential option, resource option and zero length variable option and payload. Hop option keyed hash is HMAC-SHA-1 [32] on the receiving and sending side. Credential used in example case is X.509 based certificate with RSA encryption with MD5 hash signature. V3 X.509 extensions were used to encode user attributes and VE and service identifiers. Signature in the credential option, computed on originating node, was RSA encryption with SHA-1 hash, RSA key length used was 768 bits. Certification path length was one.

On the left hand side of Figure 5-4 is a case where user credential is contained in the packet and is validated on the node together with all other checks explained in section 5.8. Hop part represents costs of validating hop option on receiving side and building a new hop option on sending side. Encoding/decoding represents costs of decoding a packet and encoding it with new hop option. Other costs are related to the process of building a security context on the node, verifying user statements about VE and service Ids regarding those in credential and access control decision regarding security context of the input channel with simple policy. Signature costs are costs of validating a credential and verifying the digital signature in the packet. In this case we can pass over the node 396 packets per second.

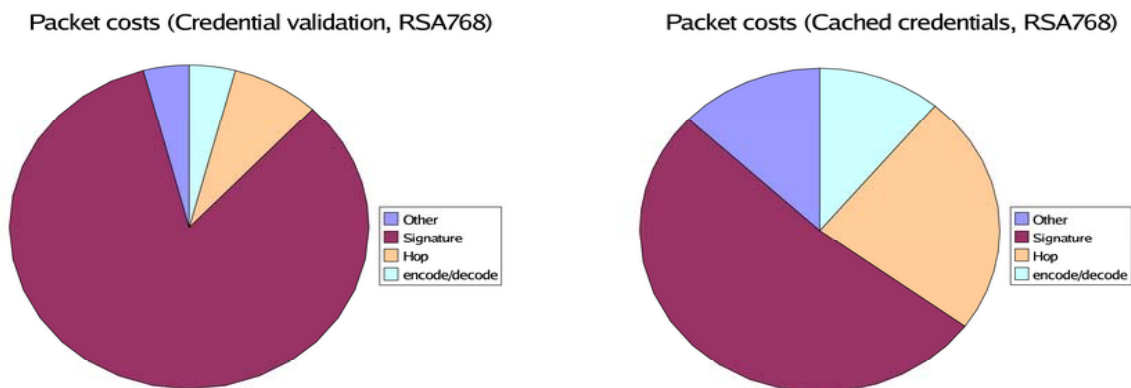


Figure 5-4: Security related packet costs

On the right hand side of the above figure, the packet related security cost of exactly the same operations is explained, but in case when the user credential is already cached on the node. Security costs are lowered for factor of three and 1190 packets per second can be passed over the node.

Packet related costs are not drastically related to the RSA key size in current setup: in the case of RSA512 the node can pass 1340 and in case of RSA1024 1003 packets per second.

Per hop costs are, as can be seen from the figure 5-4, much lower then signature related costs. As said in section 5.4.2, per hop protection also authenticate the sending node; approach is usable for inter node communication, like routing updates etc. The node can handle more than 7000 per-hop protected requests.

While the number of packets that can be passed over the node is relatively small it is hard to compare it to reported results in active networking community. The results reported are for more simple packet structures as in our case and are in range from 6000 to 11000 packets per second in similar environment reported for ANTS [6] and Bees [21]. But there were no security related cost calculated or reported. The size of the variable option has an impact on the decoding/encoding costs and per-hop protection costs, which are proportional to time needed to compute hash of the packet data. Payload size additionally increases costs to calculate the hash used to verify the digital signature in the packet.

Comparing the results for example with OpenSSL [38] library shows that the OpenSSL library is more than two times faster than the Bouncy castle library for digital signature verification (1428/3550 per sec). Using cryptographic accelerators like Broadcom based BCM5821 [39] show additional speed improvements with very light utilization of main CPU (8142 signatures verification per sec, less then

1% CPU). Using native libraries or accelerators should improve performance of security architecture regarding the active packets to few thousands per seconds.

5.10 ARCHITECTURE APPLICABILITY

The security architecture designed and implemented in FAIN should be applicable to any active networking approach. To evaluate our assumption we have applied the security architecture to the case of Active SNMP, also developed in FAIN [29].

Active SNMP is a SNAP [33] based solution for controlling and managing active node resources. SNAP is used in this solution as a carrier and a finite state machine to program a series of active network nodes through SNMP enabled network devices.

SNAP itself provides high level of safety and it even provides resource usage guarantees per packet. But the approach is as safe as much it is limited in its ability of actions that can be result of the packet evaluation on the node. In our case the evaluation of SNAP packet can result in composition with a system, that is known to have security problems [25] and the actions requested can be security critical. Every action should be authorized and the policy for the action enforced on the nodes that the packet traverse.

There are two distinct problems in integrating SNAP activator in the FAIN framework. Firstly, how to provide protection for SNAP packets while they are in-transit over the network. SNAP follows a pure active networking approach and contains intermixed code and data in the same packet. SNAP packets can legally change in the network during the evaluation on the node. Therefore, for the packet as a whole, its data origin cannot be cryptographically verified on the nodes that the packet passes. Second issue is, how to integrate SNAP daemon into the node environment and how to use FAIN based mechanisms for the authorization and policy enforcement.

SNAP packet integrity issues were tackled in the following way: after compiling a SNAP program the originating node produces a fingerprint of the program, extracting from the program the static part of SNAP packet data. Static parts are SNMP commands and related data, which will be invoked on the node during the packet evaluation. The fingerprint is stored in the ANEP packet payload while the entire SNAP packet is put into variable option. Originating node builds a credential option and digitally signs the static parts of the packet, including the fingerprint in the packet payload.

At the intermediate nodes the general scenario of the packet passing the node is exactly the same as described in the section 5.8. Packet fingerprint with known data origin is submitted to verification subsystem together with the SNAP program itself, fingerprint of the program is produced again and compared to the one that was verified. If they match, the packet will be injected in the system, otherwise it will be dropped. Verification in this way ensures that the security critical parts of the SNAP code have not changed in the network; the commands and the data, their position and occurrence is protected against unauthorized modifications.

In this way the general protection of the packet as designed in FAIN is achieved: the SNAP packets can be protected in the network with per hop protection, so only trusted nodes can process the packet, and the end-to-end authentication is possible with the data origin authentication of program fingerprint and other packet options and verification of the fingerprint.

Integration with the FAIN node services are in part covered by the integration of the SNAP daemon with the management framework. In this way the SNAP daemon is treated as any other node component also regarding security issues, e.g. of its installation, initialisation and management. SNAP daemon environment was extended on the node with a trap system that intercepts SNAP packet requests and invokes actions on the node corresponding to these requests. Additionally two helper components were designed and implemented that take care of resubmitting and intercepting the packets going in and coming from SNAP daemon. Those two components also take care of synchronizing the SNAP packet evaluation with its security context, built from the active packet external representation. In this way, SNAP packet actions can be authorized in the general way, as described in section 5.7.2 SNAP packet security context is compared with a security context of the trap system and authorization decision enforced.

5.11 EVALUATION OF THE SECURITY ARCHITECTURE

Security architecture was evaluated through number of properties, namely flexibility, security, reliability, performance and scalability.

We were addressing flexibility in many parts of security architecture design: addressing the problem from general point of view so the architecture should be applicable to all active networking approaches, even existing one as shown in section 5.10. With the choice of unidirectional authentication with digital signature mechanism and public key cryptography and design of credential option supporting multiple type of credentials it is possible to support different trust management approaches. Multiple credentials related options are possible in a single packet so described relationships can be even broader, spanning multiple domains. On the domain borders it is possible to replace user credentials by trusted services with domain credentials, which can be used in other domains. User involvement in the security can be made minimal; regarding the service, its parameters and business model, it should be enough that they are negotiated in advance and suitable credentials issued regarding the user initiated key pair. Access to, in our case to user secure store, should be enough to enable user to access the service and use it in accordance with the contract negotiated. Architecture can be used in transport, management and control plane; though its usage can be limited because of the performance issues in transport plane, as we will discuss later.

The choice to build the security architecture at the NodeOS layer and transparent to the services enhances reliability of the system. Architecture is designed and implemented once instead of multiple times for different protocols or on different system layers. Similar is true for integration of the component model and security. All system components are treated equally regarding the security issues for the installation, initialisation, termination and runtime operations. Besides RCF can guarantee reserved share of the computational and communicational resources as discussed in section 5.4.6 which helps the system to operate reliable even when node resources are scarce. Separation of VEs and services also adds to reliability of a system as a whole: if a service in a VE is possibly compromised this fact should affect this service only and in no way other services in the same or in other VEs.

Performance evaluation as presented in section 5.10 shows, that initial performance should be sufficient for applications in management and control plane; performance is not adequate for transport plane without improvements in software coded and used, cryptographic accelerators or trade offs.

From scalability point of view the mechanisms selected, namely per hop protection and end-to-end protection, scale well. Per hop protection can handle sufficient number of neighbour nodes and requests. End-to-end protection, because it is unidirectional, can be used while passing large number of nodes. The real problem is performance related because of the cost of digital signature verification. The system can support different types of credentials and trust management approaches. Multiple types of policies could be used even policies like Keynote.

From security point of view we can evaluate the presented architecture from security related high level architecture goals. Authorized use can be enforced for important system and users' resources, separation between VEs and services is enforced by the system itself. Active packets can be protected in transit; this feature is implemented only in Java based EE so the transport plane of the PromethOS EE is not supported. But both environments, PromethOS and Active SNMP with either commercial router as Hitachi GR2000 or Linux based router are managed with active packets through component based node environment and are thus secured with the same security mechanisms as the rest of the node. Verification of the code deployed to the node is in static case done with inter working with node level ASP. Dynamic verification cases can be supported as shown in the case of Active SNMP system. Common treatment is achieved through the unified component model, same mechanisms used for securing sessions and active packets, common authorization and policy enforcement based on the security context and common verification mechanism. Because of the performance issues the all three planes cannot be treated equally at the moment. With example of Active SNMP we have shown that the base security services should be sufficient and that the system can be extended with additional security mechanisms as SNAP program fingerprinting and verification.

Finally security architecture can be applied to all three type of nodes developed in FAIN; Linux router with Java based EE, PromethOS node extended with FAIN basic services and hybrid node with Hitachi GR2000 router, FAIN developed node and Active SNMP system. All three nodes can be supported by security architecture in management and control plane, transport plane support, though with low performance, is possible with Java based EE. Security architecture network experiments with two types of nodes; Linux router and hybrid node were done in FAIN setup pan-European testbed.

5.12 CONCLUSIONS

FAIN security architecture was designed as general and flexible system. Strong security on a flexible and heterogeneous network node is possible and we have achieved most of our basic goals. Through experiments we have shown that security architecture as designed and implemented can be applied to three types of nodes at least in management plane and security architecture performance should be sufficient for operations in control plane. But there is still a lot of research, implementation work and experiments that needs to be done. Node component model should be formalised together with the security architecture operations. Flat security model assumed regarding system entities in interaction should be extended to multiple entities and proper model proposed to treat them as compound principals. The security state of the node that can be exported through authorization engine should be kept on the node and continuously analysed; the same mechanism should be used for security related protocols and operation of security subsystems as verification manager, system integrity subsystem, connection manager etc. Authentication in the case of active packets has performance problems and more solutions, preserving existing flexibility, should be proposed, designed and evaluated.

6 EXECUTION ENVIRONMENTS

6.1 JAVA EE

This chapter presents the JAVA execution environment, being the implementation of the node level management layer. Details about the definitions of key interfaces can be found in appendix A.1.

6.1.1 Introduction

The JAVA execution environment is an implementation of the concept of execution environments as described in chapter 2.3. It provides a runtime support for service components implemented in JAVA [14] together with support for inter-component communication based on CORBA [13]. The runtime environment consists of a collection of generic classes and some helper classes. The generic classes can be extended by component implementations and offer an internal interface to the framework as well as call-back methods which can be overridden.

The access to the CORBA ports is secured by SSL. Clients have to provide certificates used for authentication. All interactions with a CORBA port can then be checked against policies maintained by the respective security context. Using CORBA portable interceptors and portable object adapters together with servant locators it is possible to maintain the clients' identity throughout a chain of interactions.

In particular the generic classes offered by the JAVA execution environment have been extended towards a node level management framework as presented in more detail below. The JAVA execution environment is most suitable for service components implemented in JAVA and using CORBA but can equally be used to implement wrappers for legacy systems and allows for alternative inter-component communication besides CORBA.

The domain of the JAVA execution environment is the support of portable, easy to develop service components, usually on the control or management layer but also on the transport layer for low packet rates. However, the JAVA execution environment isn't good for directly running components targeted at high-performance packet processing.

In the following the framework classes forming the JAVA execution environment and some basic services implemented therein will be presented. The framework classes implement the concepts presented in chapter 2.3. They can be used to derive specific implementations for particular service components.

6.1.2 Implementation

Figure 1 shows the hierarchy of the classes that make up the implementation of the JAVA execution environment. In the following they will be described in more detail.

6.1.2.1 Basic Component

This class implements the concept of the basic component as outlined in chapter 2.3.1. It will setup the component's initial port and provides an internal interface to the derived implementation to add or remove additional specific ports. This is done by providing a port description. Later, when clients try to get access to a specific port this description will be used to create a port entity bound to the client. There are special methods to add or remove CORBA ports. The method for the creation of a port entity can be overridden by the derived implementation in order to provide a specific handling for particular kinds of ports not based on CORBA.

The internal interface provides methods for getting the unique identifier of the component and the name of the owner. Further, there is a callback method for the initialisation of the component called by the appropriate component manager during the creation of the component. This method can be overridden when special steps have to be taken by the derived implementation. At this point the specific ports should be added.

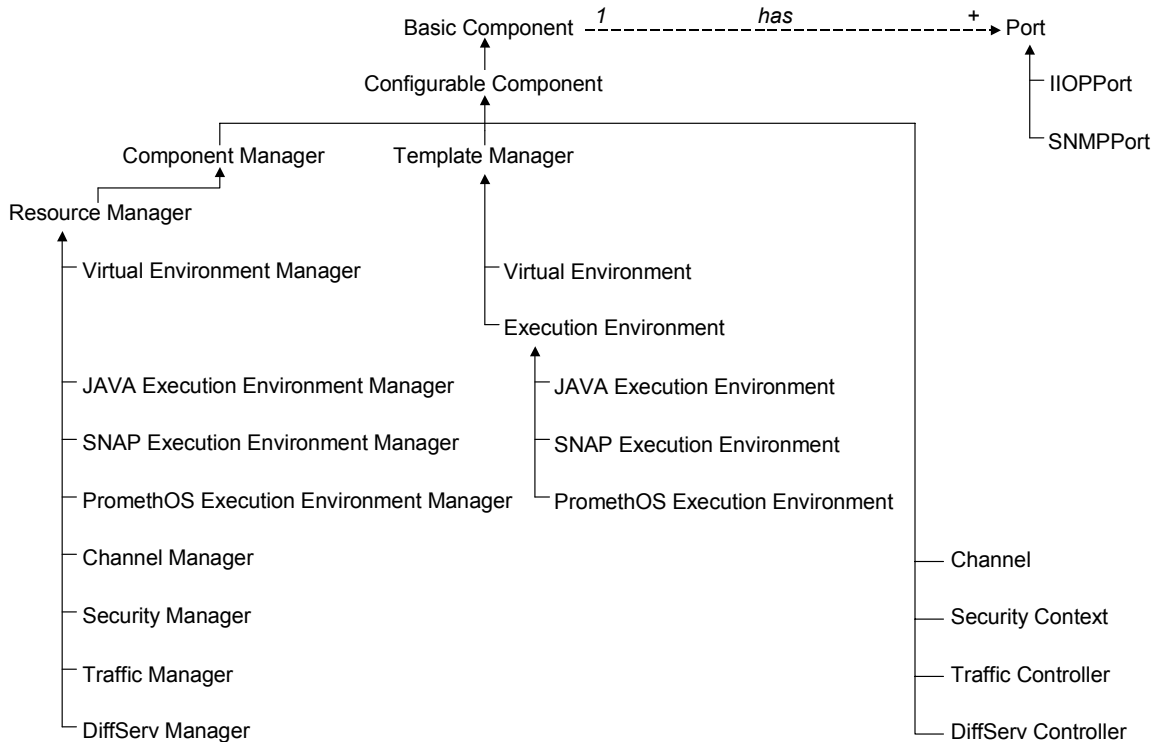


Figure 6-1: Class hierarchy for JAVA execution environment

6.1.2.2 Port

This class is a generic superclass for port classes. It defines two methods for allocating and freeing the represented port, which should be overridden by the specific subclasses. In general, when a port is allocated a reference to the port is created and an implementation is bound to the reference. In the case of a CORBA port, for example, the reference would be an IOR and the implementation a servant.

6.1.2.3 IIO Port

This class represents a CORBA port. It makes use of the portable object adapter to create port references. Together with a servant locator the references are mapped to the appropriate ports' implementations. Using a servant locator allows to perform access control prior to invoking the ports' implementations.

6.1.2.4 SNMP Port

This class represents an SNMP port. Components can use such ports to set or get values from a MIB based on an object identifier. Additionally, an SNMP port will receive traps with a defined enterprise identifier and invoke a call-back on the respective port's implementation. In contrast to the IIO port the SNMP port's reference is only valid in the same execution environment and cannot be exported.

6.1.2.5 Configurable Component

This class implements the concept of the configurable component as outlined in chapter 2.3.2. It is derived from the basic component class and additionally implements the configuration port plus the handling of properties and property observers. The internal interface provides methods for setting, getting, and changing properties as well as call-back methods for the events of changes in the state of properties.

Further, this class implements the handling of connections between ports of the represented component instance and ports of other component instances. There is a call-back method, which will be called by the framework when ports are about to be connected. This method should be overridden in order to store the address of the target port and to implement specific handling of connecting to non-CORBA ports.

6.1.2.6 Component Manager

This class implements the concept of the component manager as outlined in chapter 2.3.3. It is derived from the configurable component class and additionally implements the component manager port. Internally it cares about the management of component instances and associated profiles. It defines call-backs for the creation, activation, deactivation, and deletion of instances, which have to be overridden by the derived implementations.

6.1.2.7 Resource Manager

This class implements the concept of the resource manager as outlined in chapter 2.3.5. It is derived from the component manager class and additionally implements the resource manager port. Internally it cares about the management of supported resource dimensions and the current resource usage. Further, it sends notifications to registered clients when particular thresholds are crossed. It defines internal methods for defining supported dimensions and updating the current usage.

6.1.2.8 Virtual Environment

This class implements the concept of the virtual environment. It is derived from the configurable component class and additionally implements the template manager port as outlined in 2.3.4. Whenever a virtual environment is created via the virtual environment manager it will get references to its associated execution environments. Those references will be used to dispatch requests concerning the management of templates to the appropriate execution environment.

6.1.2.9 Virtual Environment Manager

This class is derived from the component manager class and manages instances of virtual environments. It extends the resource manager port by providing a method for retrieving instances based on the virtual network identifier.

6.1.2.10 Security Context

This class is derived from the configurable component class. It offers a port for checking the access to other ports based on the identity of the current client. It maintains policies which define particular identities and their access rights.

6.1.2.11 Security Manager

This class is derived from the component manager class and manages instances of security contexts. For more details on the internals of the security manager see chapter 5.

6.1.2.12 Execution Environment

This class implements the concept of the execution environment on a generic level. It is derived from the configurable component class and as does the virtual environment implements the template manager port. It provides the storing of templates and their descriptions; specific implementations can be derived from this class and override the call-backs for the installation and de-installation.

6.1.2.13 JAVA Execution Environment

This class is derived from the execution environment class and employs class loaders for the installation of templates.

6.1.2.14 *JAVA Execution Environment Manager*

This class is derived from the resource manager class. When a new JAVA execution environment is activated the manager will start a new process on the operating system level. The manager will monitor the CPU and memory usage of the process and care about the compliance with the environment's resource profile. In the current implementation the manager will simply kill the environment process in the case that the resource usage exceeds the defined quotas.

6.1.2.15 *PromethOS Execution Environment*

This class is derived from the execution environment class. It wraps a PromethOS kernel-space execution environment and maps requests concerning the management of templates to the PromethOS interface. Additionally there are wrapper classes for PromethOS components and respective component managers. For more details on the internals of the PromethOS execution environment see chapter 6.2.

6.1.2.16 *PromethOS Execution Environment Manager*

This class is derived from the component manager class. It simply manages instances of PromethOS execution environments.

6.1.2.17 *SNAP Execution Environment*

This class is derived from the configurable component class. It represents a SNAP daemon started as a process. Though called an execution environment it currently doesn't support the management of templates. The SNAP execution environment features the execution of active packets and uses SNMP for communication with other component instances. Thus, it is also called "active SNMP activator". For more details on the internals of the active SNMP activator see chapter 6.3.

6.1.2.18 *SNAP Execution Environment Manager*

This class is derived from the component manager class. It simply manages instances of SNAP execution environments.

6.1.2.19 *Channel*

This class is derived from the configurable component class. It is used to forward packets from the network to component instances and to take packets back and send them to the network. A channel is created per virtual environment. Particular component instances or execution environments belonging to the virtual environment can connect their ports with the ports of the respective channel. This can be done for the exchange of data packets as well as of active packets. Receiving component instances have to specify a condition based on what the channel would forward packets to them. Currently the channel class supports ports for CORBA communication and plain UDP sockets.

6.1.2.20 *Channel Manager*

This class is derived from the component manager class and manages channel instances. It uses operating system specific means (e.g. Linux Netfilter) to intercept packets from the forwarding path and dispatches them to a channel instance with a matching condition. For more details on the internals of the demultiplexing done by the channel manager see chapter 4.

6.1.2.21 *DiffServ Controller*

This class is derived from the configurable component class and offers a port for configuring packet handling based on the model of differentiated services [10].

6.1.2.22 *DiffServ Manager*

This class is derived from the component manager class and manages instances of diffserv controllers. In the current implementation it demonstrates how to wrap the functionality provided by a legacy router (Hitachi GigabitRouter2000) [11]. In order to receive configuration requests this class offers an SNMP based port for communication with the SNAP execution environment. For more details on the internals of the diffserv manager see chapter 3.

6.1.2.23 *Traffic Controller*

This class is derived from the configurable component class and offers a port for configuring packet handling based on different queuing models.

6.1.2.24 *Traffic Manager*

This class is derived from the component manager class and manages instances of traffic controllers. It also demonstrates how to wrap functionality of a legacy system but instead of using a hardware router it makes use of the Linux traffic control [12]. For more details on the internals of the traffic manager see chapter 3.

6.1.3 Use Cases

The prototype implementation of the JAVA execution environment and the integrated basic services and wrappers is used in various scenarios for demonstration purposes, in particular

- a video-on-demand scenario demonstrating the interworking of the management layer and the PromethOS execution environment;
- a web-TV scenario demonstrating the interworking of the management layer, the demultiplexing, and active service components;
- a differentiated services scenario demonstrating the interworking of active packets executed in the SNAP execution environment, the management layer, and the diffserv controller running in the JAVA execution environment interfacing with legacy router hardware.

Some basic use cases will be informally presented here and are parts of the aforementioned scenarios.

6.1.3.1 *Booting the Management Layer*

Booting the management layer is done by starting the privileged virtual environment and installing the basic services like management of environments, security, demultiplexing channels, traffic, etc. The privileged virtual environment will then publish the reference to its initial port on a well know TCP port.

6.1.3.2 *Creating a Virtual Environment*

A virtual environment is created by the node owner for a service provider so that the latter can deploy services for its customers on the node. After getting the reference to the privileged virtual environment's initial port the template manager port is accessed. There the initial port of the virtual environment manager is retrieved and the component manager port is accessed. At this port a new virtual environment is created and activated.

When a virtual environment is created a resource profile has to be specified. The profile defines all required resources to be attached to the new virtual environment. The virtual environment manager will contact the respective resource managers to create and activate the required resources.

6.1.3.3 *Deploying a Service*

Typically, a service provider will deploy services in its virtual environments and make them available to its customers. Deploying is done by getting a reference to the initial port of the service provider's virtual environment. To get this reference the virtual environment manager has to be contacted like described in the previous use case.

Then the template manager port of the virtual environment is used to install one or more service components, which involve(s) the instantiation of the respective component managers. Later the component managers will be used to create and activate instances of the service components. The initial configuration of a service is done by accessing the individual components and setting their properties and interconnecting them.

6.1.4 Conclusion

This chapter presented a collection of classes that make up the JAVA execution environment. This environment provides support for service components implemented in JAVA and using CORBA or SNMP for communication. Further, it is extensible for other types of communication; for example, the channel class implements ports based on plain UDP sockets for other components to connect to.


This collection of classes is used to implement the node level management layer based on the requirements and concepts as defined in chapter 2. Additionally, it serves as an extensible framework allowing to derive specific service components without having to care about aspects like access control, resource monitoring, component deployment and activation, etc. For environments, which aren't implemented in JAVA, for example a hardware router or a kernel-based environment, wrappers can be used to map the respective abstractions into the node level management layer.

This chapter also presented some basic use cases, which are part of the bigger demonstration scenarios.

6.2 PROMETHOS EE

In the past, the functionality of routers was very limited, namely forwarding packets based on the destination address. Recently, new network protocols and extensions to existing protocols have been proposed and implemented, requiring new functionality in modern routers at an increasingly rapid pace. However, present day commercially available routers typically employ a monolithic, closed architecture, which is not easily upgradeable and extensible to keep up with new innovations.

When PromethOS has been designed, the following requirements were set before in order to overcome these limitations:

-  Modularity. The router architecture is designed in a modular fashion with components coming as plugins, which are modules that are dynamically loaded into the kernel and have full kernel access without crossing address spaces. Dynamical loading and installation of plugins into the operating system kernel at run-time allows for flexible code development and deployment. Plugins are code modules that implement specific router functionality. For example, a router plugin might implement encryption functionality.
- Flexibility. For each plugin class, multiple plugin instances can be created. Different configurations of the same plugin can co-exist simultaneously in the kernel, with plugin instances sharing the same code but operating on their own data. Plugins may be instantiated as often as required. An instance is a specific run-time configuration of a singular plugin. Quite often, it is required to have several plugin instances of one plugin in the kernel, e.g. packet scheduling. There, a packet scheduler may work in different configuration, hence different instances, for several interfaces. State-of-the-art packet schedulers are configured hierarchically. Quite often, the several modules are used which work in different hierarchical levels. At different levels, the instances of one plugin may be configured differently.
- Interfaces. A consistent and simple interface must be provided such that the plugins may easily be programmed. This also includes the reaction of plugins to signals. PromethOS provides a standard set of signals in order to provide interoperability of plugins.
- Packet classification. By defining filters, incoming data packets are classified to belong to a data flow and by binding a plugin instance to such a flow, all matching packets will be processed by the corresponding plugin instance. Efficient mapping of specified flows and the possibility of binding flows to specific plugin instances are required. Usually, filters specify sets of flows. For example, a filter may classify a TCP data flow from network 172.16.7.0/24 to the host 172.16.0.65. Filters may classify packets according to end-to-end application flows. A 6-tuple may specify filters: <Source Address, Destination Address, Protocol, Source Port, Destination Port, Interface>. Every element of a 6-tuple may be specified as irrelevant. For the former example, a filter is specified as: <172.16.7.0/24, 172.16.0.65/32, TCP, *, *, *>. Obviously, a filter for end-to-end application flows needs a filter specification according to the aggregate level: a single flow requires the specification of all parameters.
- Performance. An efficient data path is guaranteed by implementing the complete data path in kernel, preventing costly context switches, and by using efficient mechanisms for packet classification.
- Integration in Linux. The implementation needs only minimal changes to the existing Linux source code and can easily be integrated into newer releases.

We have implemented our framework based on the Linux 2.4 kernel. We have selected this platform because of its portability, freely available source code, extensively documented, wide-spread use as a state-of-the-art experimental platform by other research groups and by its continuously growing acceptance in industry. Due to its modularity and extensibility, we are convinced that our proposed framework makes it a useful tool for researchers in the field of programmable router architectures and protocol design.

6.2.1 Architectural Overview

A PromethOS node consists of components running in the Linux kernel space and those running in the Linux user space. In kernel space, data path service components for efficient packet processing are located, i.e. PromethOS plugins that act on data of every or nearly every packet. In user space, interfaces to the management framework (Virtual Environment Manager, VEM) are located.

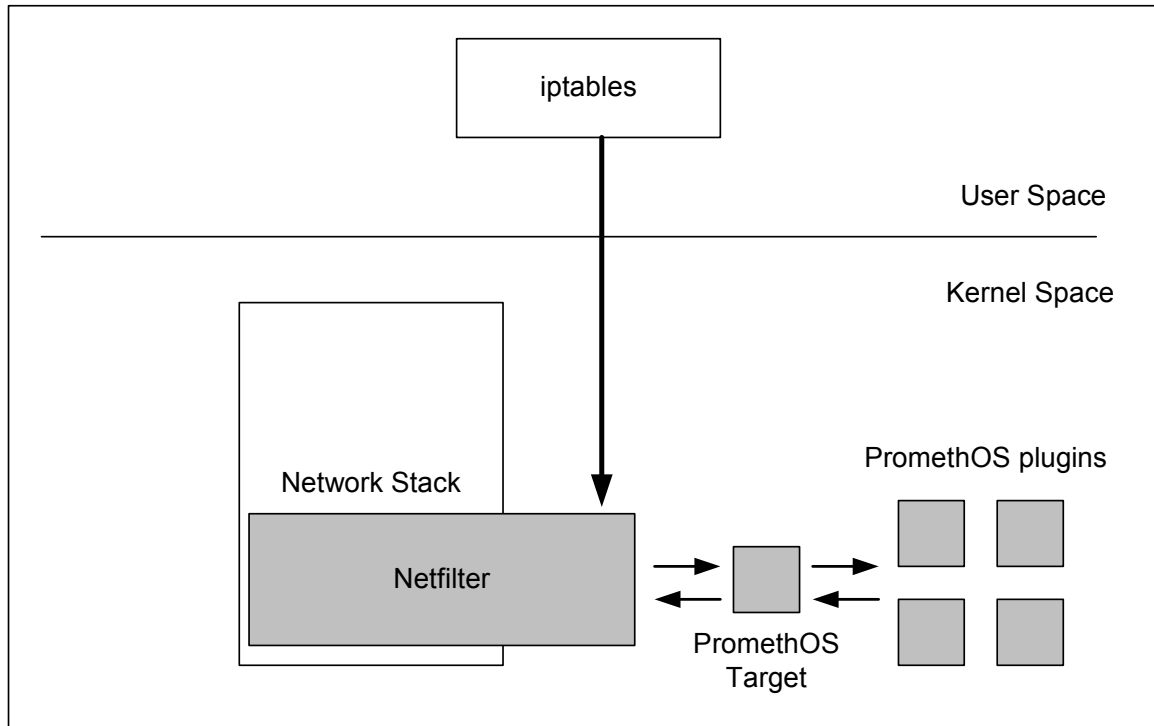


Figure 6-2: Netfilter and PromethOS

The main objective of our proposed architecture is to build a modular and extensible networking subsystem that enables to deploy and configure packet processing components (PromethOS plugins) for specific flows. Figure 6-2 illustrates our dynamically extensible router architecture.

The most important components are as follows:

- Netfilter classifies packets according to filter rules at various hooks. Packets matching a filter are passed to registered kernel modules for further processing. To control kernel space environments, user space tools must be provided. In Linux, the user space tool iptables is the standard way for handling Netfilter kernel space functionality.
- The plugin framework provides an environment for the dynamic loading of plugin classes, the creation of plugin instances as well as their configuration and execution.
- The plugin loader is responsible for requesting plugins from remote code servers which store plugin classes in a distributed plugin database.
- The PromethOS User Space Library interfaces the PromethOS environment to the Virtual Environment Manager.

6.2.2 Netfilter Framework

The netfilter framework [rusty_russel:netfilter] provides flexible packet filtering mechanisms which are performed at various hooks inside the network stack. Kernel modules register callback functions that get invoked every time a packet passes the respective hook. The user space tool iptables allows defining rules that are evaluated at each hook. A packet that matches these rules is handed to the target

kernel module for further processing. The netfilter framework together with the iptables tool provide the minimum mechanisms required to load modules into the kernel, specifying packet matching rules evaluated at hooks, and the invocation of the matching target module.

However, netfilter has a serious restriction since all loadable modules must be known at compile time to guarantee proper kernel symbol resolution for the linking process. Thus, only kernel modules that have been statically configured can be loaded into the networking subsystem. This is a significant limitation since we envision a router architecture that allows loading arbitrary new components at runtime.

6.2.2.1 Netfilter Architecture

Netfilter, developed and implemented by Rusty Russell, provides a framework for packet filtering. Every protocol supported by Netfilter specifies several hooks, for example, IPv4 defines 5 hooks. A hook allows the interception of packet flows along the kernel internal packet path. Kernel extensions may register at one or several of these hooks. Netfilter calls these extensions every time a packet arrives at a hook. Such an extension may inspect the packet, modify it, ask Netfilter to accept it or to drop it or to enqueue it for user space. Figure 6-3 provides an overview of the Netfilter framework for IPv4. Netfilter is documented in detail on <http://netfilter.samba.org>.

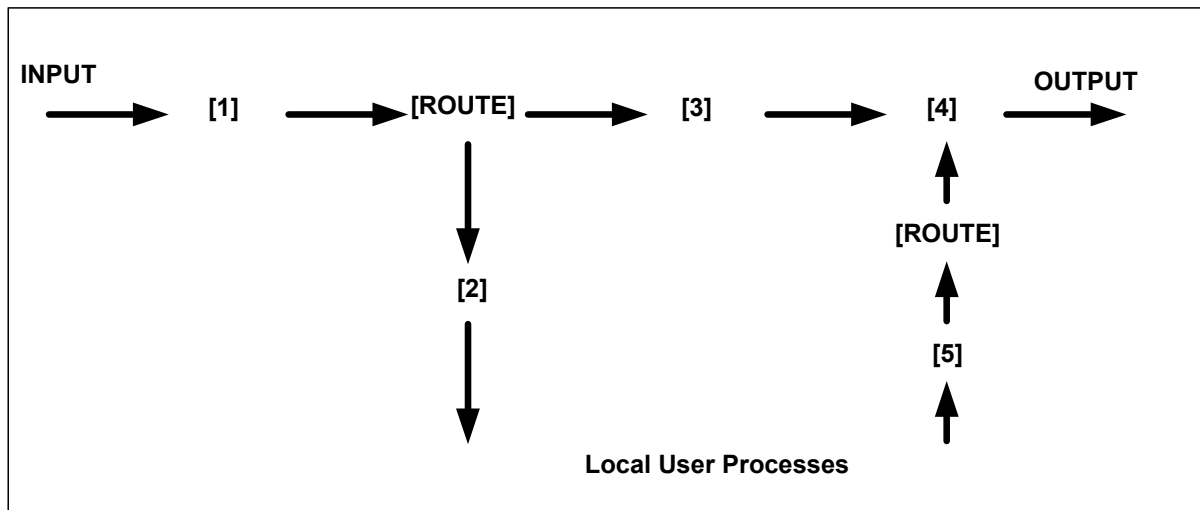


Figure 6-3: Netfilter Architecture for IPv4

6.2.2.2 Extensions to Netfilter

As a basis, Netfilter provides a framework to which kernel extensions may be bound. However, Netfilter requires the extensions to be available at compile time, i.e. kernel extensions must be specifically compiled per kernel build. Therefore, in order to implement the above mentioned requirement “flexibility”, a suitable extension is needed which provides the kernel space to install plugins at runtime without the need to be specifically built for one kernel only. This is what PromethOS provides, i.e. mechanisms to load plugins and bind them to specific flows at runtime. An overview of Netfilter together with PromethOS is provided in Figure 6-2.

The following functionality has been implemented:

- A new Netfilter table into which filter expressions may be specified. These filters may bind plugins bound to flows.

- A new Netfilter target³. By this target, plugins are managed and controlled at run-time. This target dispatches packets to the appropriate plugin instances and classifies packets for flows.
- iptables as the user space tool has been extended such that the new arguments can be passed to the new Netfilter target.
- Control functionality is provided to query statistical information.

6.2.3 PromethOS Netfilter-Table

To easily separate PromethOS flows from normal filters, and to allow the filters to be hooked to every available hook, a new Netfilter-table, promethos, is implemented. This table registers during load time at the Netfilter framework.

PromethOS plugins should be able to be activated at every hook in the Netfilter framework. Therefore, the table must register at every hook. However, this pre-registering consumes resources during run-time: every hook gets run for every packet. So, to optimise, the superfluous hooks are to be removed.

The PromethOS Netfilter-table is implemented as a single Linux kernel module. At module initialisation-time, this module registers at the appropriate places in the Netfilter framework. Figure 6-4 provides an overview with the example of an IPv4 protocol-hook configuration.

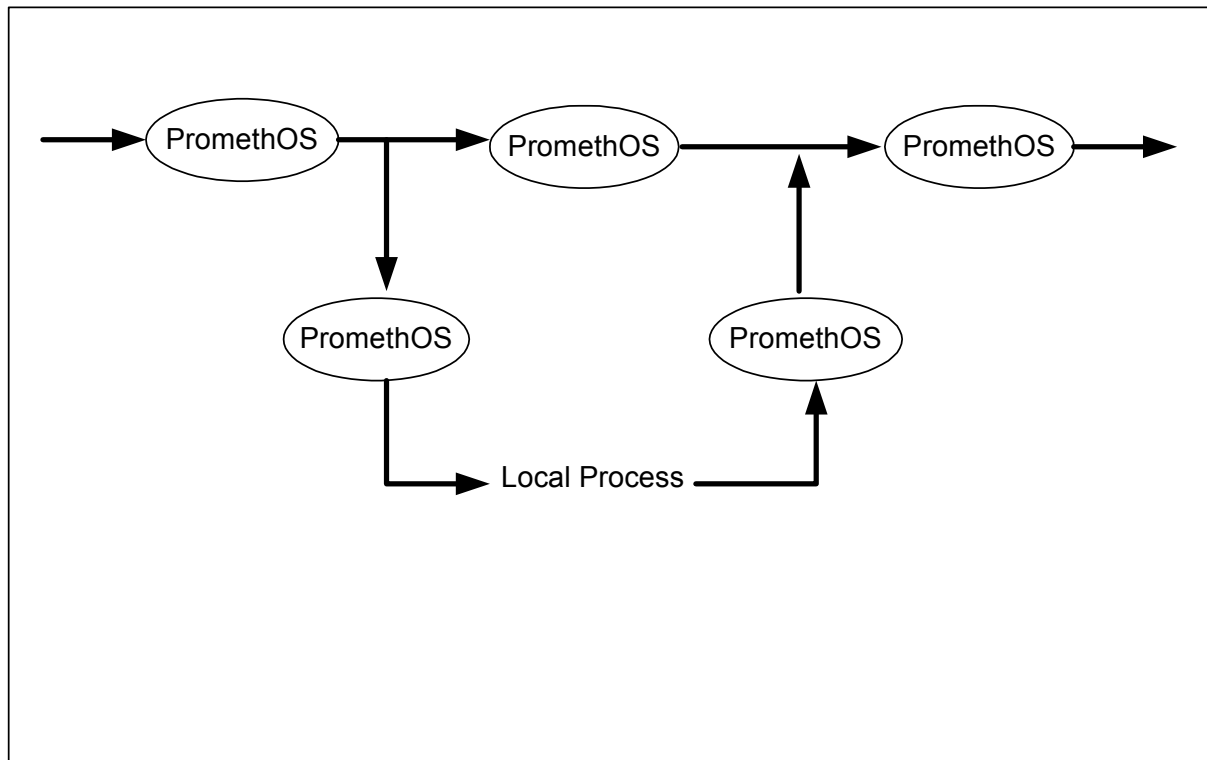


Figure 6-4: PromethOS Netfilter-Table Hooks

Flow entries in the PromethOS table point to a specific Netfilter-Target, which is named PROMETHOS. This target provides the data structures that are necessary for the management of PromethOS plugins. Control functionality is provided that keeps track of the loaded plugins and their instances.

³ The Netfilter-target provides the PromethOS framework. The PromethOS framework controls packet dispatching and plugin installation.

For this issue, the target exports two interfaces. One is used at initialisation time of the plugin to register; the other one is called at the time a plugin stops and gets unregistered. By these mechanisms, PromethOS is aware of the status of loaded plugins.

6.2.4 Plugin Framework and Execution Environment

The plugin framework manages all loadable plugins and dispatches incoming packets to plugins according to matching filters. When a plugin initially gets loaded into the kernel, it registers its virtual functions with the plugin framework. Once a packet arrives and needs to be processed by a plugin, the framework invokes the previously registered plugin-specific callback function. Since plugins register their entry-points, the entry functions do not need to be known at compile time, and for this reason the plugin framework can load and link any plugin into the kernel. Every PromethOS plugin offers an input and output channel (in accordance with [40]) representing a control and reporting port. The control port is used for managing the PromethOS plugin (such as configuration); the reporting port is read-only to collect status information from the plugin.

6.2.4.1 Plugin Classes and Instances

For the design of plugins, we follow an object-oriented approach. A plugin class is a dynamically loadable Linux kernel module that specifies the general behavior by defining how it is initialized, configured, and how packets need to be processed. A plugin instance is a runtime configuration of a plugin class bound to a specific flow. An instance is identified by a node unique instance identifier. In general, it is desirable to have multiple configurations of a plugin, each having its own data segment for internal state. Multiple plugin instances can be bound to one flow, and multiple flows can be bound to a single instance. Through a virtual function table, each plugin class responds to a standardized set of methods to initialize, configure, reconfigure itself, and for processing packets. All code is encapsulated in the plugin itself, thus the plugin framework is not required to know anything about a plugin's internal details. Once a packet is associated with a plugin, the plugin framework invokes the processing method of the corresponding plugin, passing it the current instance (data segment) and a pointer to the kernel structure representing the packet (struct sk_buff).

6.2.4.2 Control from User Space

PromethOS and its plugins are managed at load-time by providing configuration parameters and at run-time through the control interfaces via the /proc file system.

When the PromethOS plugin framework initially gets loaded, it creates the entry /proc/promethos. Below this entry, the control and reporting ports of individual plugins are registered. PromethOS plugins are loaded by iptables which we extended with semantics required for the PromethOS plugin framework.

The communication to control plugins and report messages between user space and plugins follows a request-reply approach. A control message is addressed to the appropriate plugin by passing the plugin instance identifier as a parameter and the plugin then responds with a reply. The PromethOS framework interfaces between the user space management tools and the PromethOS plugin. When the proc file system is accessed for reading, the framework calls the print() function of the plugin to retrieve the information provided by the user. If the /proc file system is accessed for writing, the PromethOS framework passes the information received to reconfig() function of the PromethOS plugin.

6.2.5 PromethOS User Space Library

The PromethOS User Space Library deals with interfacing the VEM to the PromethOS internal management framework that is responsible for creating kernel space Execution Environments (EE), loading and unloading of PromethOS plugins as well as inserting them into the chain of plugins for creating the data path service. The PromethOS User Space Library is based on the iptables as it has been extended for PromethOS as well as on the interfaces created by the PromethOS kernel Framework, i.e. it connects to these interfaces and transmits the control and management requests by

VEM.

6.2.6 Summary, Outlook and further work

PromethOS provides a high-performance PromethOS EE in the Linux 2.4 kernel space. The NodeOS functionality by PromethOS manages the creation and instantiation of the Virtual Environment object, the PromethOS EE and of the PromethOS plugins. The VE is used for resource management. Assigned to the VE, several PromethOS EE instances can be created. These EEs are attached to the hooks provided by the PromethOS Netfilter table, which in turn attaches itself to the hooks provided by the Netfilter framework. Inside an EE, PromethOS plugins are chained to create the active service. A PromethOS plugin must have one input port, may export a control interface, and may provide one to several output ports. These ports are used to allow for service-internal dispatching to different strings of modules.

The PromethOS User Space Library encapsulates the complexity of managing PromethOS in a library that can be linked to control applications as it has been done with the VEM in the context of the Integrated Video on Demand scenario.

Currently, Resource Control is carried out only in terms of rudimentary accounting. Further research on in-kernel space resource constraint enforcing need to be carried out to allow foreign, untrusted code to be safely installed and run on a PromethOS node.

6.3 ACTIVE SNMP ACTIVATOR

6.3.1 Introduction to SNAP EE

This section describes a new service control mechanism using SNMP across a network for controlling and managing service activities within and around FAIN active nodes. In our approach, once a VE is given the authority to access the requested network resources, services can be provided by any SNMP-enabled network devices e.g. FAIN active nodes. Finite state machines are implemented by active packets; these machines can then program a series of SNMP-enabled network devices in a synchronised manner, and provide a means for rollback: should any request for a network resource fail, the fulfilled requests made earlier are released. Using such kind of active packet mechanism, it will be possible to implement complex network reconfigurations; for instance, it can create IPSec tunnels and modify routing table entries to use it. The system uses the Safe Network with Active Packets (SNAP) programming language to implement the finite state machines. It offers the facilities to issue SNMP commands that can be applied to network devices. Integration of SNAP Activator with Security and DeMUX is provided for by the ANEP-SNAP Packet Engine (ASPE) (see later sections for details).

6.3.2 System Design Goals

6.3.2.1 Interceptor Paradigm

Active network management is the application area for the Active SNMP Activator (also known as the SNAP Activator) system. Active networking is an interceptor paradigm. It is difficult to develop applications that rely upon intercepting data packets because the interceptor must decode each data packet and its intention must be understood. In contrast to traditional network management, what is needed for effective network management is an in-band management capability. Each flow will negotiate its next hop before it goes there. It will be seen that SNAP and SNMP can come close to achieving this: the SNAP packet will precede the data and go to the next hop, it will then establish a route for the data that will follow it. The information used by the SNAP packet to choose the route will state the intention of the data flow. For example:

- The data flow may be an HTTP request for a large resource to be delivered to the requesting machine.
- The data flow may be the start of a large system backup: sending large amounts of data to the accepting machine.

In both cases, the data flow will be asymmetric; in the former case, it will require a larger capacity in the reverse direction; in the latter, in the forward direction. The information that states the requesting machine's intent is only available at the edge of the network where the request is made - only the local network administration knows the capability and priority of its machines for a limited resource. The statement of intent is contained in an active packet that attempts to match its source with the sink of the data flow. The active packet can revise and choose how the source and sink impedances are matched.

6.3.2.2 Active Packets & Active Extension Technology: SNAP & SNMP

SNAP is a programming language that provides active packets at high level of safety. Essentially, SNAP packets are UDP packets with assembly codes embedded. As a SNAP packet traverses through the network, simple computations⁴ such as to add and remove data to a stack within the packet can be performed. This is a genuinely active mode of operation. It will be seen that the application of SNAP within active network management is as a finite state machine that follows the progression of a reconfiguration of a network. Finite state machines do not need a complex runtime environment and

⁴ SNAP programming language is an assembly language and it cannot perform any computations that are comparable in complexity to that of a C or Java program; nor can it support the wide range of data types that are available in these languages.

SNAP will prove to be sufficient. SNMP has been chosen as the active extension technology to work with SNAP for a number of reasons:

- It is the de-facto language of network management.
- SNMP v.3 provides cryptographically strong role-based access control.
- An extensible MIB and programmable SNMP v.3 agent have become available for conventional operating systems.
- Machines that run conventional OS are now capable to act as network routers as well.

The extensible MIB allows complex operations to be simplified to one macro instruction. In SNMP, the GET and SET commands can be thought of as operation codes for a programming language: LOAD and STORE. One could think of the object identifiers in the extensible MIB as memory locations. We have written simple programs in SNAP for testing the operational state and branching to different operation sequences in the Diffserv Scenario.

6.3.2.3 SNAP Packet Format

1	2	3	4
IP Destination			
IP Source			
Resource Bound		Port	
Entry Point		Code Size	
Heap Size		Stack Size	
Code (~ bytes)			
Heap (~bytes)			
Stack (~bytes)			

Figure 6-5: SNAP Packet Format

- **Resource Bound**

Similar to the use of hop count, the resource bound limits the number of hops that a SNAP packet can propagate.

- **Port**

The port to which this particular SNAP packet should be delivered (default: 7777).

- **Entry Point**

Indicates at which instruction execution is to begin.

- **The Code, Heap and Stack**

The Code field keeps a sequence of bytecode instructions; the Heap keeps large values such as byte arrays and / or tuples (an array of small integers); the Stack keeps small values such as integers or addresses.

6.3.3 System Design

6.3.3.1 Injectors

The SNAP Injector injects SNAP programs into the network to reconfigure network devices. The SNAP Injector will decide whether or not to inject code once it has intercepted a request for a data flow from its own network. The Injector intercepts and interprets some part of an application protocol. For example, the injector may intercept NFS (Network File System) requests, obtain the user

identification contained within the NFS request and use that to priorities the use of bandwidth to deliver data. It can also make use of the MAC addresses, the IP addresses, and the current network topology in its own administrative domain. In effect, it monitors the state of its own network and its connection with external networks. When a new network condition develops, an injector will attach control information to the data flows it hopes to control:

- *Appearing flows* - A new network condition is engendered by a new data flow and the control information will be attached to the new flow.
- *Disappearing flows* - An injector may know that a flow, or a set of flows has finished: a machine or user or another network may have disappeared from the network.

The SNAP Injector in SNAP Activator will inject code. SNAP code will be sent to the same host as the data that triggered the network event. All active SNAP-enabled routers will intercept these packets as they traverse the network. As a general rule, SNAP packets should precede the data packets in the network, so that the data packets will not be able to traverse the network until the SNAP packets have a created a route for them.

6.3.3.2 Interceptors

Intercepting SNAP packets is more complicated than injecting them. These are the constraints: 1) the code has to be executed as quickly as possible, so that the packet can be quickly forwarded and minimise latency during the establishment of the data flow; 2) the functionality required will need to make use of active extensions on the node; 3) active extensions require blocked I/O; 4) blocked I/O cannot be performed in the same thread as the execution of the SNAP packet, because it would add too much latency. SNAP packet interception is performed at the SNAP daemon and the SNAP Interceptor.

6.3.3.3 Active Extensions

SNAP provides a facility to access services within the SNAPD: *CALLS* (call service). A service is a C function. This will be used to dispatch the SNMP commands embedded in the SNAP program. SNAP also provides a facility to read variables maintained by the SNAPD: *SVCV* (service variable collect). This will be used to return the state of SNMP variables. In this way, an SNMP command can be issued on one thread and the result can be returned, stored within the SNAPD and dispatched as the result in a subsequent SNAP packet.

6.3.3.4 System Architecture

Interceptors will be SNAPD running on active routers, they will listen on several SNAP control ports. At the time of writing, the SNAP interpreters are not part of a system that has packet spooling, which is still an experimental of the Linux kernel [11]. Figure 6-6 presents the architecture of SNAP Activator.

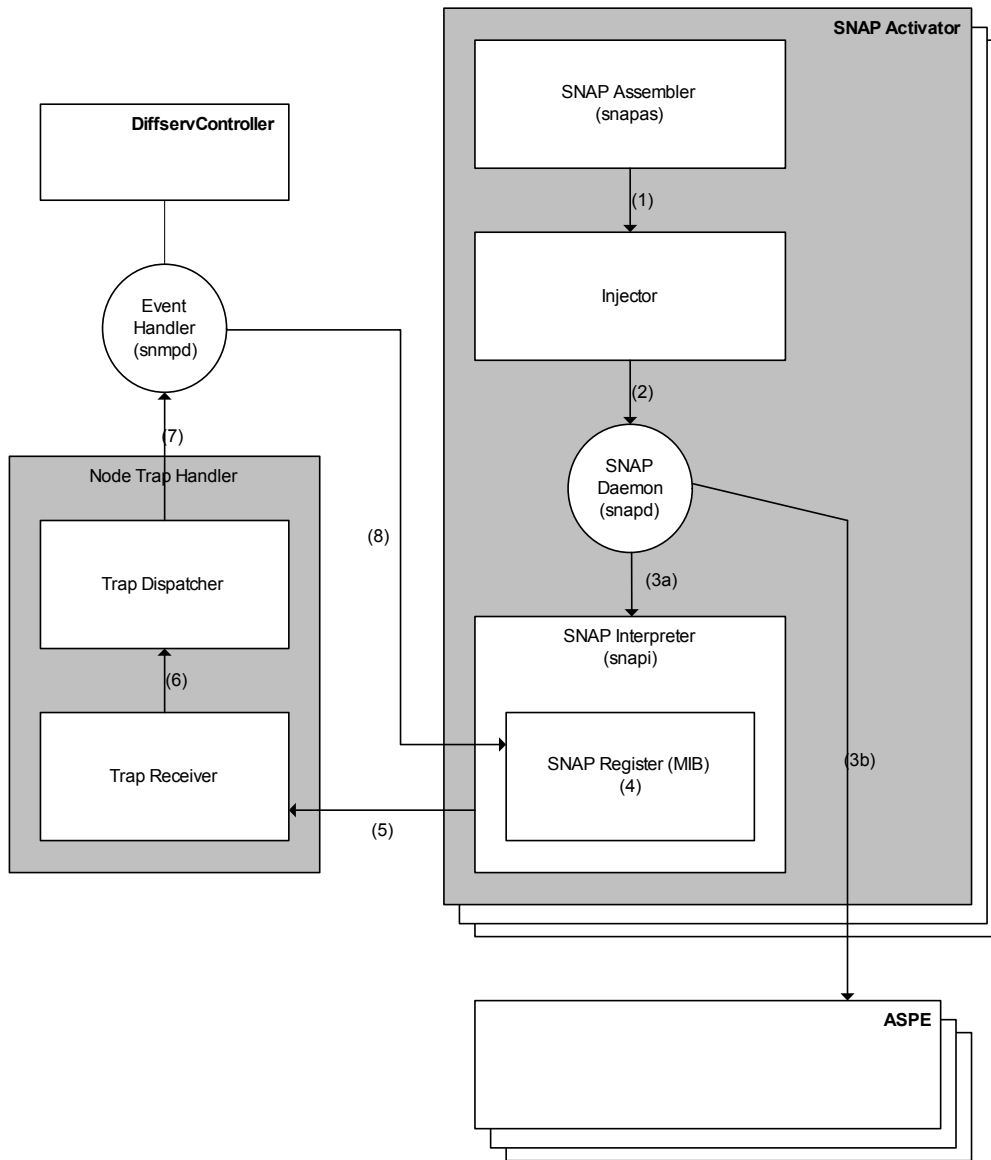


Figure 6-6: SNAP Activator Block Diagram

SNAP packet flow: (1) the SNAP Assembler takes a SNAP program written in SNAP assembly language, and produces a wire-format binary SNAP packet; (2) binary SNAP packets are injected to the SNAP daemon by the SNAP Injector; (3a) if the SNAP packet's destination matches with the local node address, the SNAP daemon will simply forward the packet to the local SNAP Interpreter for packet processing; (3b) else if the SNAP packet's destination is elsewhere (i.e. destination address does not match with the local address), the SNAP packet will be processed by the SNAP daemon in the same way as stated in (3a), additionally the SNAP daemon will generate a copy of this SNAP packet (that will be forwarded to the ASPE for ANEP encapsulation), the copy will then be forwarded to its next hop via the ANEP-SNAP Packet Engine (ASPE) and DeMUX (see later sections for details); (4) the SNAP code of the SNAP packet is executed at the SNAP Interpreter, local values will be set in the SNAP Register; we have implemented this SNAP Register as a MIB for the Diffserv Scenario, as a consequence object identifiers (OID) will be set in this MIB; (5) a trap is raised and will be captured by the Node Trap Handler; (6) the Trap Receiver passes the trap to the Trap Dispatcher; (7) the Trap Dispatcher is responsible for dispatching traps to their corresponding Event Handler; (8) the Event Handler receives the trap and reads the value (e.g. the OID) that was set in the SNAP Register, desired action based on the value that was set in the Register will be performed by the Event Handler.

6.3.4 Introduction to the ANEP-SNAP Packet Engine (ASPE)

6.3.4.1 Active Network Authentication Challenges

In conventional networks, one of the major focuses in authentication is to authenticate a client whilst he or she is requesting for some services. Such type of authentication is defined to be end-to-end authentication: an end user – who is requesting for a service – must be authenticated by a remote server or by an intermediate active node. Authentication in active networks should include end-to-end authentication as well as hop-to-hop authentication: active packets must be authenticated at the recipient i.e. an intermediate active node or a remote server etc. based on the identity of the node on which the packets were last modified. Both end-to-end and hop-to-hop authentication are needed in active networks due to the dynamic nature of active packets [5]: active packets that carry code are injected for the purpose of service control, and these active packets are then intercepted and their embedded codes are executed at some intermediate active nodes (i.e. the modifying nodes) before reaching the desired destination e.g. a remote server. At each modifying node, the results of code execution may be added to the active packet before the packet is forwarded to the next hop. So, the contents of an active packet may vary at each modifying node whilst the packet is in-transit across different domain. Thus, end-to-end and hop-to-hop authentication (as well as node and link integrity protection) are needed in active networks; otherwise intermediate active nodes might process active packets that were originated from spoofed nodes or have been contaminated by spoofed intermediate nodes on the network.

6.3.4.2 Existing Solutions

1) Traditional Authentication Techniques

Traditional authentication techniques or existing systems that include authentication mechanisms such as Kerberos, IP security protocol (IPSEC), transport layer security (TLS) etc. involve either or a combination of or all of the knowledge, possession, and biometrics authentication factors. These techniques are generally considered to be sufficient for protecting the end-to-end authenticity of network entities (e.g. clients, nodes etc.) in conventional networks. However, as these techniques were not originally developed for hop-to-hop authentication they are insufficient for active network authentication: for instance, should these traditional authentication techniques be employed in a hop-to-hop manner: each hop would have to authenticate active packets arriving from its neighbour; but then unless all intermediate nodes can be trusted, the end-to-end authenticity of the active packets cannot be covered [5].

2) PLAN: A Packet Language for Active Networks

Hicks who developed a packet language for active networks (PLAN) [7] at the Computer and Information Science (CIS) Department at University of Pennsylvania (UPENN) had recommended the use of cryptographic techniques for active packets authentication. There are two types of cryptographic techniques for authentication: symmetric and asymmetric. Symmetric authentication requires every modifying node to share a key for signing active packets once modifications have been applied; this is not feasible unless all nodes are trusted on a network. Asymmetric authentication uses a private key for signing, but as the source's private key would always be kept locally on the source node, the modifying nodes would not be able to reproduce the source's signature after modifications have been applied. Alternatively, each node can sign packets by using its own private key on the modifications it has made on the packet; but then the old signature i.e. the source's signature that was generated by the source may be overwritten; moreover, digitally signing each of the modifications on each active packet at each modifying node may generate an undesirable performance overhead on intermediate modifying nodes⁵.

3) SANTS: Secure Active Node Transfer System

⁵ Digital signatures are expensive to compute: a 500 MHz Pentium can generate 100 1024-bit RSA signatures per second. Also, a 1024-bit RSA signature would take up 128 byte, which is the same size as a standard SNAPPING active packet program [3][4].

Murphy and others has proposed to use digital signatures as well as credential references to protect the end-to-end and hop-to-hop authenticity of active packets (and their clients) in secure active node transfer system (SANTS) [5][6]. In their approach, active node transfer system (ANTS) [6] packets are encapsulated into active network encapsulation protocol (ANEP). The ANEP format is modified in SANTS: the ANEP Payload field is separated into a static and a variable area for keeping static (i.e. the MD5 hash identifier of the active codes) and variable (i.e. network resource bound) data respectively.

There are several technical issues must be resolved before the type of approach proposed by Murphy could become practical.: ANEP packet header format has to be modified in SANTS in order to keep a list of credential references of the modifying nodes, as a result active nodes must therefore be re-configured in order to recognise these variable length packets at packet interception; thus it is important to address how active node would handle packets of variable length. Also, the splitting of active packets must be performed efficiently: in the SANTS approach, they would have to analyse an ANTS packet, then separate the contents of the packet into static and dynamic parts before encapsulating these parts separately into ANEP. Their approach may generate a significant performance overhead should the process (of analysing ANTS packets, deciding which parts of the packets are static and which parts are dynamic, then splitting the packets and re-unifying the packets at a recipient node) is repeated at every single modifying node for every single ANTS packet.

6.3.4.3 Authentication in the SNAP EE

As discussed in earlier chapters, SNAP [4] developed by some developers at UPENN is used as the active packet language in SNAP Activator (also known as Active SNMP Activator). SNAP Activator generates SNAP packet programs that carry various SNMP commands. SNAP packet programs are to be executed on FAIN active nodes for the purpose of service control. It should be noted that, as SNAP is designed to be a light-weight and simple active packet language, SNAP itself provides no facility for authentication at all. UPENN claims that a SNAP packet program is safe to execute without even examining it: given that SNAP packet program does not contain any primitives to exert control over local nodes or other packets [4]; it is therefore important to provide the necessary security facilities to protect the authenticity of SNAP packet programs in FAIN. To integrate with DeMUX and to solve the security threats of SNAP, SNAP packet programs that are generated by SNAP Activator are encapsulated by ANEP. ANEP-encapsulation of SNAP packet programs is performed at the ANEP-SNAP Packet Engine (ASPE). The SNAP-encapsulated ANEP packets are known as ANEP-SNAP packets.

6.3.5 Requirements of the ASPE

As a general rule, a SNAP packet must provide the following information to the ASPE: a) the VE ID that the SNAP packet owns, b) the EE ID, c) its destination address and d) the SNAP packet ID; also Security should provide a Security ID (SID) for each processed (secured) SNAP packet.

1) End-to-End Protection

Recalling from the previous discussion on the authentication challenges faced by authentication systems in active networks, in order to enforce both end-to-end and hop-to-hop authenticity, we propose to determine the static data of SNAP packet programs, and then to encapsulate these data in a separate field of ANEP. The ASPE examines the contents of SNAP packet programs and encapsulates the static contents of a SNAP packet into the payload of an ANEP packet. We define the SNMP commands that are carried in SNAP packet programs to be the static data, whereas the dynamic data are those that are being kept in SNAP packet programs' stack and heap. Note that these static data are generated at the source (i.e. a SP which represents a client) and will not be modified whilst the packet is in transit (e.g. the same SNMP command will be executed at each traversing node). As a consequence, this field will be digitally signed by the source's private key (the digitally signing is performed at SEC), and the signature will be verified by each of the intermediate modifying nodes as well as by the remote server at the desired destination. In such way the end-to-end authenticity of the packet program would be covered.

2) Hop-to-Hop Protection

The entire of a SNAP packet program is encapsulated into the ANEP Option 5. Note that the data in the stack and the heap of SNAP are to be modified at each hop (e.g. the results of code execution at each modifying node and the current node address may be push back onto the stack), also the packet's network resource bound may be decremented. A SHA hash code is generated for the SNAP packet and is appended to Option 5. When an ANEP-SNAP packet arrives at a node, the SNAP packet will be extracted from Option 5, and a hash code of the extracted SNAP packet will be generated. This new hash code will be compared to the hash code that is appended to Option 5 to ensure the integrity of the SNAP packet that is currently in-transit across different domains. In such way we would be able to provide hop-to-hop protection to SNAP packets. As discussed earlier, cryptographic techniques are not suitable for protecting the authenticity of the dynamic data of active packets; as a result, SEC will enforce integrity protection on the packet to protect the packet from illegal modifications whilst the packet is in transit.

6.3.6 ANEP-SNAP Packet Format

An ANEP-SNAP packet has the following format:

N	4N + 0 Byte	4N + 1 Byte	4N + 2 Byte	4N + 3 Byte
0	Version	Flags	TYPE_ID	
1	ANEP_HL		ANEP_PL	
2	FLG_OP_TYPE_1		OP_LENGTH_1	
3	VE_ID (4 bytes)			
4	FLG_OP_TYPE_2		OP_LENGTH_2	
5	EE_ID (4 bytes)			
6	FLG_OP_TYPE_3		OP_LENGTH_3	
7	SCHEME_ID (4 bytes)			
8	DST_IP (4 bytes)			
9	FLG_OP_TYPE_4		OP_LENGTH_4	
10	SID (8 bytes)			
11				
12	FLG_OP_TYPE_5		OP_LENGTH_5	
13	SNAP Packet (~ bytes)			
m				
m+1- m+21	SNAP Packet Hash Code (20 bytes)			
M+22	PAYLOAD_LENGTH		SNAP Packet Static Command (~ bytes)	
n				

Figure 6-7: ANEP-SNAP Packet Format

- **FLG_OP_TYPE_5**

The flag for SNAP packets (Option 5) is 1005.

- **OP_LENGTH_5**

The length of a SNAP packet is variable, but is always within the 4-byte boundary.

- **SNAP Packet Hash Code**

A 20-byte hash code of the SNAP packet.

- **PAYLOAD_LENGTH**

The length of the SNAP static command.

6.3.7 System Architecture

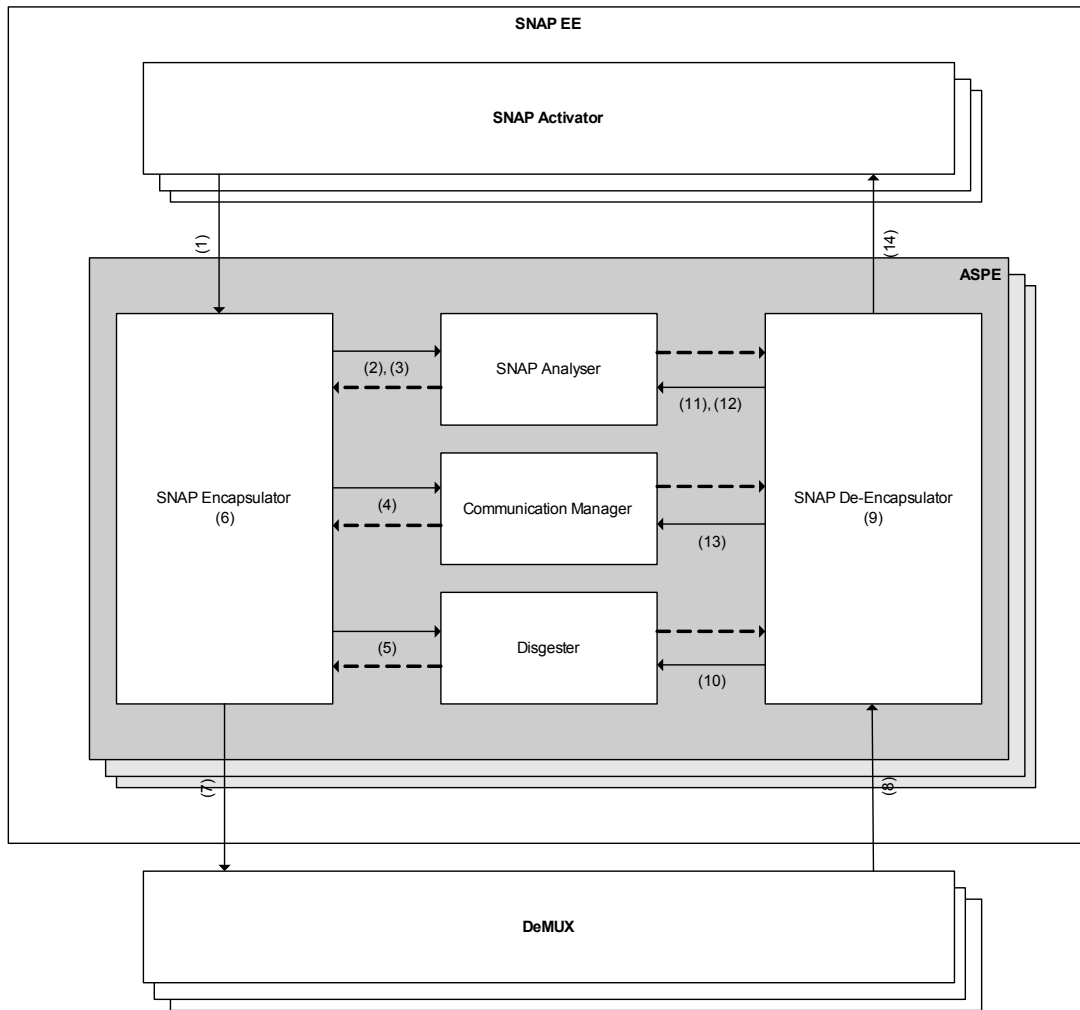


Figure 6-8: Block Diagram for ANEP-SNAP Packet Flow

ANEP-SNAP packet flow: (1) SNAP Activator generates a SNAP packet; (2)&(3) the SNAP Analyser determines the VE ID, EE ID, destination address, SNAP Packet ID and the SNAP static command from the SNAP packet; (4) the Communication Manager uses the SNAP Packet ID as a reference to extract the corresponding Security ID (SID) of this SNAP packet from its database, if no SID is found then this SNAP packet will be treated as a fresh packet, as a result Option 4 will be set to zero; (5) the Digester generates a hash code for this SNAP packet; (6) the SNAP Encapsulator encapsulates the analysed SNAP packet (and its hash code) and its static command into ANEP Option 5 and the Payload field respectively, the SNAP Encapsulator then assigns the VE ID, EE ID, destination address and the SID of this SNAP packet to Option 1 to 4 respectively.

When an ANEP-SNAP packet arrives at its next hop, (9) DeMUX dispatches the packet to the SNAP De-Encapsulator (after successful security checks), the SNAP De-Encapsulator extracts the SNAP packet (and its hash code) from Option 5; (10) the Digester calculates a hash code for the SNAP packet, and matches this recently generated hash code with the hash code that is extracted from Option 5; if the two hash codes match then the packet's integrity has not been compromised whilst the packet was in-transit; (11)&(12) if the destination of the ANEP-SNAP packet is not local then the SNAP Analyser will extract the SNAP packet ID and the Security ID from the SNAP packet and Option 4 respectively, (13) the Communication Manager keeps this ID-SID pair in its database, (the ID pair is needed by the SNAP Encapsulator for ANEP-SNAP encapsulation - see step 4); (14) the SNAP De-Encapsulator passes the SNAP packet to SNAP Activator. The same process is repeated at every traversing node.

6.3.8 Conclusion

We have outlined in this section the underlying concepts and the fundamental requirements of SNAP Activator and ASPE. The interceptor paradigm of SNAP Activator makes it an ideal application for service control in FAIN active networks. However, as SNAP is designed to be a light-weight protocol additional security measurements must be applied to SNAP systems before SNAP can be used as a practical active packet language for service control among FAIN active routers. We have addressed the needs to enforce hop-to-hop authentication as well as end-to-end authentication in active networks due to the dynamic nature of SNAP active packets. Hop-to-hop authentication should be applied to the dynamic data of active packets and is defined as: each recipient to authenticate the received active packets based on where the packets were last modified; whereas end-to-end authentication should be applied to the static data of active packets and is defined as: the recipient to authenticate the received active packets based on the source's identifiers on the packets. Digital signatures are used for protecting the end-to-end authenticity of the static parts of active packets; whereas hop-to-hop authenticity is enforced by integrity protection. The integrity and authenticity of SNAP packets are protected by ASPE and Security Manager. We have described the architecture of SNAP Activator and ASPE in this section, we have described the interaction between SNAP Activator and ASPE with other FAIN components such as DeMUX and SEC by illustrating the data and control path between each component.

7 CONCLUSIONS

The FAIN Active Node architecture introduced an entirely new concept; the combination and coordination of different Execution Environments that represent different technologies which are then used to host service components and interact with each other as part of the overall service operation. This has been achieved through the definition of a reference model that combines EEs, VEs, and service components.

In order to realise this reference model an object oriented management framework has been proposed that provides a number of classes with methods that allows EEs to be deployed in VEs, in turn, service components to be deployed and linked with existing services in EEs and exports control interfaces of these components for their configuration. The operation of the EEs, and in turn of the services that running in them, is regulated by the FAIN resource control framework based on the VE abstraction which is used as a principal for accounting and resource allocation and partition. Moreover all the operations take place in a secure environment that has been created based on the flexible FAIN security architecture, which provides authentication, authorisation of the use of resources and verification of packets.

The FAIN node architecture and its components have been implemented and a number of different EEs residing in different operational planes and interworking with each other have been created the functionality and mechanisms of which achieve the design goals. The JavaEE, residing in the management plane, binds together all the FAIN node components as well as the other EEs. The PromethOS EE, residing in the transport plane, acts as a high performance EE and supports component-based architectures. Finally, we have created an Active SNMP that resides in the control plane and is used to control the behaviour of services that may exist in other EEs.

All the aforementioned functionality has been deployed, evaluated and tested in the FAIN testbed by means of a number of practical scenarios. The scenarios that originated from this workpackage were the Diffserv scenario, the Video on Demand scenario and the Web Service Distribution scenario. Their detailed description may be found in D9.

8 REFERENCES

- [1] D. Scott Alexander, Bob Braden, Carl A. Gunter, Alden W. Jackson, Angelos D. Keromytis, Gary J. Minden, and David Wetherall. Active network encapsulation protocol (anep). Active Network Group draft, July 1997. <http://www.cis.upenn.edu/~switchware/ANEP/>.
- [2] S. Denazis, T. Suzuki, T. Becker, D. Gabrielcic, A. Lazanakis et al., Revised Active Network Architecture and Design,” FAIN Deliverable 4, May 2002
- [3] D. X. Song, A. Perrig, “Advanced and Authenticated Marking Schemes for IP Traceback”, INFOCOM 2001, Vol. 2, pp. 878-886.
- [4] SNAP (Safe and Nimble Active Packets) <http://www.cis.upenn.edu/~dsl/SNAP/>
- [5] S. Murphy, “Strong Security for Active Networks”, IEEE OpenArch 2001
- [6] D. J. Wetherall, “ANTS: a toolkit for building and dynamically deploying network protocols”, OpenArch 1998, San Francisco, CA, April 1998, pp.117-129, IEEE.
- [7] M. Hicks, “A Secure PLAN”, IWAN 1999, July 1999, vol.1653
- [8] D. Raz, “An Active Network Approach for Efficient Network Management” IWAN99, July 99, <http://www.cs.bell-labs.com/who/ABLE/>
- [9] Ch. Garbrecht, C. Klein, A. Galis et al., “Requirements Analysis & Overall AN Architecture”, FAIN Deliverable 1, May 2001
- [10] S. Blake, D. Black, M. Carlson et al., “An Architecture for Differentiated Services”, IETF RFC 2475, December 1998, <http://www.ietf.org/rfc/rfc2475.txt>
- [11] Hitachi Internetworking, <http://www.internetworking.hitachi.com>
- [12] B. Hubert et al., “Linux Advanced Routing & Traffic Control HOWTO”, <http://lartc.org>
- [13] Object Management Group, “Common Object Request Broker Architecture”, http://www.omg.org/technology/documents/corba_spec_catalog.htm
- [14] Sun, <http://java.sun.com>
- [15] Bert Hubert “Linux Advanced Routing & Traffic Control HOWTO”
- [16] S. Floyd, V. Jacobson “Link-sharing and Resource Management Models for Packet Networks“ IEEE/ACM Transactions on Networking, Vol. 3 No. 4, August 1995
- [17] D. Scott Alexander, Kostas G. Anagnostakis, William A. Arbaugh, Angelos D. Keromytis, and Jonathan M. Smith. The price of safety in an active network. Technical Report Penn Technical Report MS-CIS-99-04, University of Pennsylvania, February 1999.
- [18] ITU-T X.509 (2000) | ISO/IEC 9594-8:2000 - information technology - open systems interconnection -the directory: Public-key and attribute certificate frameworks. Final Draft International Standard, June 2000.
- [19] D. Scott Alexander, William A. Arbaugh, Angelos D. Keromytis, and Jonathan M. Smith. Security in active networks. In *Secure Internet Programming: Issues in Distributed and Mobile Object Systems*, Lecture Notes in Computer Science State-of-the-Art. Springer-Verlag, 2000.
- [20] Active Networks Security Working Group. Security Architecture for Active Nets, May 2001.
- [21] Tim Stack, Eric Eide, and Jay Lepreau. Bees: A secure, resource-controlled, java-based execution environment, December 2002.
- [22] FAIN team. Active node architecture and design. FAIN Public Deliverable D2, May 2001.
- [23] M. Wahl, T. Howes, and S. Kille. RFC 2251: Lightweight directory access protocol (v3). Standards Track, December 1997.

- [24] Matt Blaze, Joan Feigenbaum, John Ioannidis, and Angelos D. Keromytis. RFC 2704: The KeyNote trust-management system, version 2, September 1999.
- [25] CERT. Multiple vulnerabilities in many implementations of the simple network management protocol. Advisory CA-2002-03, February 2002.
- [26] Dan Decasper, Guru Parulkar, Sumi Choi, John DeHart, Tilman Wolf, and Bernhard Plattner. A scalable high performance active network node, October 1998.
- [27] DoD Trusted computer system evaluation criteria. DoD Standard, December 1985.
- [28] D. Eastlake and O. Gudmundsson. RFC2538:storing certificates in the domain name system (dns). Standards Track, March 1999.
- [29] Walter Eaves, Lawrence Cheng, Alex Galis, Thomas Becker, Toshiaki Suzuki, Spyros Denazis, and Chiho Kitahara. SNAP based resource control for active networks. In *IEEE GLOBECOM 2002 Proceedings*, November 2002.
- [30] C. Ellison, B. Frantz, B. Lampson, R. Rivest, B. Thomas, and T. Ylonen. RFC 2693: SPKI certificate theory, September 1999. Network Working Group.
- [31] Michael Hicks and Angelos D. Keromytis. A secure PLAN. In *Proceedings of International Workshop for Active Networks (IWAN) 1999*, pages 307-314, June/July 1999.
- [32] H. Krawczyk, M. Bellare, and R. Canetti. HMAC: Keyed-hashing for message authentication. RFC2104, Informational, February 1997.
- [33] Jonathan T. Moore, Michael Hicks, and Scott Nettles. Practical programmable packets. In *INFOCOM 2001 proceedings*, April 2001.
- [34] George Ciprian Necula. *Compiling with proofs*. PhD thesis, School of computer science, Carnegie Mellon University, September 1998.
- [35] Murphy S., Lewis E., Puga R., Watson R., and Yee R. Strong security for active networks. In *IEEE OPENARCH 2001 Proceedings*, April 2001.
- [36] B. Schneier. *Applied Cryptography: Protocols, Algorithms, and Source Code in C*. John Wiley and Sons, Inc., second edition, 1996.
- [37] R. Shirey. RFC 2828: Internet security glossary, May 2000. Internet Engineering Task Force.
- [38] OpenSSL, <http://www.openssl.org>
- [39] Broadcom based cryptographic accelerator, <http://www.broadcom.com/products/5821.html>
- [40] L. Peterson et al., Active Network Working Group, NodeOS Interface Specification.
- [41] FAIN Project Deliverables – www.ist-fain.org/deliverables
- D1-Requirements Analysis & Overall AN Architecture
 - D2-Initial Active Network and Active Node Architecture
 - D3-Initial Specification of Case Study Systems
 - D4-Revised Active Network Architecture and Design
 - D5-Specification of Revised Case Study Systems
 - D6-Definition of evaluation criteria and plan for the trial
 - D14-FAIN Overview
 - D8-Final Specification of Case Study Systems
 - D9-Evaluation Results and Recommendations

- D40-FAIN Demonstrators and Scenarios

APPENDIX A

A.1 INTERFACE DEFINITIONS OF THE JAVA EXECUTION ENVIRONMENT

This chapter features definitions of the key interfaces (also known as CORBA ports) of the JAVA execution environment. Since the JAVA execution environment is used to execute the node level management layer it also exhibits the details of the component model.

A.1.1 Identification

All components are identified by a unique name. Further they get a security identifier, which is mapped, to a particular context by the security manager. When a client wants to interact with a component it first has to get access to the desired port and the client has to authenticate itself. This can be done implicitly using certificate information from the underlying SSL layer or explicitly by specifying the client's identity and credentials.

```
// IDL
typedef string tInstanceID;
typedef sequence <tInstanceID> tInstanceIDList;
typedef long long tSID;
typedef sequence <octet> tOctetList;
struct tCredential {
    string name;
    tOctetList key;
};
typedef sequence <tCredential> tCredentialList;
typedef string tIdentityName;
struct tIdentity {
    tIdentityName name;
    tCredentialList credentials;
};
```

A.1.2 Properties

Components may have a number of properties representing their configuration. A property is a pair of a name and a value.

```
// IDL
typedef string tPropertyName;
typedef sequence <tPropertyName> tPropertyNameList;
struct tProperty {
    tPropertyName name;
    any value;
};
typedef sequence <tProperty> tPropertyList;
```

A.1.3 Ports

Components are accessed and interconnected by ports. Ports can be used for exchanging information or to model this exchange. Ports are also useful to express dependencies among components. A port is created when a client requests access to it. Thus, a component's implementation can keep track of who is accessing which port.

A port has an identifier valid in the context of the holding component as well as a reference to the component itself. A port is described by its kind (i.e. actor or reactor), an address (i.e. the endpoint for data exchange like an IP address, a memory address, an IOR, etc.), a format (i.e. the protocol used for data exchange like IP, ATM, IIOP, HTTP, etc.), and a name.

```
// IDL
typedef string tPortFormat;
typedef string tPortType;
typedef string tPortName;
typedef string tPortID;
typedef any tPortAddress;
enum tPortKind {
    Actor,
    Reactor
};
struct tPortDescription {
    tPortKind kind;
    tPortFormat format;
    tPortType type;
    tPortName name;
    tPropertyList capabilities;
};
typedef sequence <tPortDescription> tPortDescriptionList;
struct tPort {
    tPortID id;
    tInstanceID component;
    tPortAddress address;
    tPropertyList details;
    tPortDescription description;
};
typedef sequence <tPort> tPortList;
```

A.1.4 Interface iComponentInitial

At the initial port of a component its name, security identifier, and supported ports can be retrieved. In order to get access to a port the client has to authenticate itself. There is also an operation to explicitly end the usage of a port.

```
// IDL
interface iComponentInitial {
    tInstanceID getID ();
    tSID getSID ();
    tPortDescriptionList getPortDescriptions (
        in tIdentity who
    );
    tPort accessPort (
        in tPortName name,
        in tIdentity who
    );
    tPort accessPortWithDetails (
        in tPortName name,
        in tIdentity who,
        in tPropertyList details
    );
    void loosePort (
        in tPortID id,
        in tIdentity who
    );
};
```


A.1.5 Interface iConfiguration

A configurable component exposes a port for configuration issues. At this port it is possible to get and set properties as well as to register for property observation. The name and owner of the component can be set or retrieved.

Additionally, the ports of a component can be connected to other ports. Before connecting a port a client has to get access to it. There are methods for getting previously access ports, for connecting and disconnecting ports, and for getting ports connected to a specific port.

A component can be started and stopped. This is particularly useful for moving components: first stop the component, then get the component's properties, delete the component, re-instantiate the component at a different location, set the previously retrieved properties, and finally start the new component.

```
// IDL
interface iConfiguration {
    void init (
        in tInstanceID id,
        in tIdentity owner,
        in tPropertyList setup
    );
    void addProperty (in tProperty property);
    void removeProperty (in string name);
    tProperty getProperty (in string name);
    tPropertyList getPropertyMatch (in string prefix);
    void changeProperty (in tProperty property);
    void addOrChangeProperty (in tProperty property);
    tPropertyList getState ();
    void setState (in tPropertyList properties);
    void registerPropertyObserver (
        in iConfigurationObserver observer,
        in tPropertyNameList names
    );
    void deregisterPropertyObserver (
        in iConfigurationObserver observer,
        in tPropertyNameList names
    );
    void suspend ();
    void resume ();
    boolean isSuspended ();
    void setID (in tInstanceID id);
    void setOwner (in tIdentity owner);
    tIdentity getOwner ();
    tPortList getPorts ();
    tPort getPortByName (in tPortName name);
    tPortList getConnectedPorts (
        in tPortID port
    );
    void connectPort (
        in tPortID origin,
        in tPort target
    );
    void disconnectPort (
        in tPortID origin,
        in tPort target
    );
};
```

```
};
```

A.1.6 Interface iConfigurationObserver

The configuration of a component may be observed by interested clients. An observer has to implement call-back operations in order to get notifications about changes of specific properties or the set of properties as a whole. The call-back interface can be registered at the respective component.

```
// IDL
interface iConfigurationObserver {
    oneway void propertyChanged (
        in tProperty old,
        in tProperty new
    );
    oneway void propertyAdded (in tProperty new);
    oneway void propertyRemoved (in tProperty old);
};
```

A.1.7 Interface iTemplateManager

A template manager manages templates (e.g. JAVA classes, object files, etc.) for component instances. This comprises the installation, removal, and updating of templates. A template manager is implemented by a virtual environment representing a specific service provider or by an execution environment. A special template manager is implemented by the privileged virtual environment representing the node provider.

When installing a new template on a node a list of properties describing the template has to be provided. The description contains:

- the identifier of the virtual environment in which the service should be installed, i.e. the identity of the according provider,
- the identifier of the execution environment type in which the service should be installed,
- the path to the code archive,
- the entry point of the component manager acting as a factory for instances of the installed template, and
- more service specific information to be used by the component manager.

The template manager will instantiate the component manager and initialise it with the given property list. The component manager can now use an execution environment specific API to reflect the installation inside the execution environment. As a result of the installation procedure a name identifying the template will be returned. This name can be used to get the reference to the respective component manager in order to create instances of the service. Further, there is an operation to get the template's description given the template's name.

When a template is uninstalled the corresponding component manager has to be destroyed. It is specific to the service whether running component instances are destroyed, too. When a template is updated the corresponding component manager has to be re-instantiated. Whether this affects running component instances is again service specific.

```
// IDL
typedef string tTemplateLocation;
typedef string tTemplateID;
typedef sequence <tTemplateID> tTemplateIDList;
struct tTemplateDescription {
    string name;
    string version;
```

```
    string mainClassName;
    string mainCodePath;
    string eeIdentifier;
    string vnIdentifier;
    tPropertyList properties;
};
typedef sequence <tTemplateDescription>
    tTemplateDescriptionList;
interface iTemplateManager {
    tTemplateID install (in tTemplateDescription description);
    tTemplateID installSetOwner (
        in tTemplateDescription description,
        in tIdentity owner
    );
    void deinstall (in tTemplateID id);
    string getReference (in tTemplateID id);
    iComponentManager getManager (in tTemplateID id);
    iComponentInitial getManagerInitial (in tTemplateID id);
    tTemplateDescriptionList getDescriptions ();
    tTemplateDescription describe (in tTemplateID id);
    tTemplateIDList findByDescription (
        in tTemplateDescription description
    );
    void update (
        in tTemplateID id,
        in tTemplateDescription description
    );
};
```

A.1.8 Interface iComponentManager

A component manager is used to manage instances of components. This comprises the creation, activation, discovery, deactivation, deletion, and update of instances. For the creation of an instance a resource profile has to be specified. This profile defines the instance's resource requirements and is used by the manager to check the availability of resources.

When resources are available and an instance was created it may be activated. During activation the instance will be initialised by the component manager. The owner will be set to the identity of the caller as determined during authentication. The name and setup properties are passed on to the created component. The component will initially be in stopped state.

Updating is useful if a new template was installed. During an update a component will be stopped and its state as represented by its properties will be saved. Then a new instance will be created from the updated template, initialized with the previously saved properties, and finally started.

```
// IDL
struct tComponent {
    tInstanceID id;
    iComponentInitial initial;
};
typedef sequence <tComponent> tComponentList;
interface iComponentManager {
    tInstanceID createInstance (in tPropertyList profile);
    tInstanceID createInstanceSetOwner (
        in tPropertyList profile,
        in tIdentity owner
    );
};
```

```
void deleteInstance (in tInstanceID id);
void activateInstance (
    in tInstanceID id,
    in tPropertyList setup
);
void deactivateInstance (in tInstanceID id);
tPropertyList dumpInstance (in tInstanceID id);
void suspendInstance (in tInstanceID id);
void resumeInstance (in tInstanceID id);
boolean isInstanceSuspended (in tInstanceID id);
string getInstanceReference (in tInstanceID id);
iComponentInitial getInstanceInitial (in tInstanceID id);
tComponentList getInstances ();
tComponentList getInstancesByOwner (in tIdentity owner);
void updateInstances (in tInstanceIDList ids);
void updateAllInstances ();
};
```

A.1.9 Interface iResourceManager

A resource may have different dimensions (e.g. CPU time and memory for processing) and there may be multiple units for a particular dimension (e.g. kilobytes, megabytes, etc.). The usage of a resource is expressed as a sequence of triplets including the dimension, the unit, and the actual value.

A resource manager is a special component manager. It creates and deletes resources, i.e. instances with associated resource quotas. In general it is possible that resources offer specific ports, e.g. a file resource could offer operations for reading and writing. However, there may be resources without any port, in this case the resource manager just manages names and returns a null reference as initial interface in the resource's component description.

Resource managers additionally provide operations for getting the supported dimensions and units as well as for converting between units. When a resource is created properties are used to define the desired resource quotas. The quota supported parameters are specific to the managed resource and part of the specification of a resource manager.

```
// IDL
typedef string tUnit;
typedef sequence <tUnit> tUnitList;
typedef string tDimensionName;
typedef sequence <tDimensionName> tDimensionNameList;
struct tDimension {
    tDimensionName name;
    tUnitList units;
};
typedef sequence <tDimension> tDimensionList;
struct tUsage {
    tDimensionName dimension;
    tUnit unit;
    double value;
};
typedef sequence <tUsage> tUsageList;
interface iResourceManager : iComponentManager {
    tUnitList getUnits (in tDimensionName name);
    tDimensionNameList getDimensionNames ();
    tDimensionList getDimensions ();
    tUsage convert (in tUsage usage, in tUnit newUnit);
};
```

A.1.10 Interface iResourceMonitor

A resource manager may offer a port for monitoring created resources and the overall availability of the managed resource. Monitoring can be done by polling or by registering a call-back interface for getting notifications when a threshold is reached.

A previously created resource can be monitored by specifying the resource's name. A special name is supported to represent the amount of unused managed resources. There are upper and lower thresholds, which will raise a notification when reached from below or above, respectively. A hysteresis value can be specified to prevent unwanted flooding with notifications when the usage is varying around the threshold. For upper thresholds a notification will only be sent when the current value dropped below the mark minus the hysteresis value and raised again above the mark. For lower thresholds a notification will only be sent when the current value raised above the mark plus the hysteresis value and dropped again below the mark.

```
// IDL
typedef string tThresholdID;
interface iResourceMonitor {
    tInstanceID getRemainderID ();
    tUsageList getCurrentUsage (
        in tInstanceID resource,
        in tDimensionNameList dimensions
    );
    tThresholdID addUpperThreshold (
        in tInstanceID resource,
        in tUsage mark,
        in double hysteresis,
        in iResourceObserver observer
    );
    tThresholdID addLowerThreshold (
        in tInstanceID resource,
        in tUsage mark,
        in double hysteresis,
        in iResourceObserver observer
    );
    void adjustThreshold (
        in tThresholdID thresholdID,
        in double mark,
        in double hysteresis
    );
    void deleteThreshold (in tThresholdID thresholdID);
};
```

A.1.11 Interface iResourceObserver

Clients may have interest in the current usage of a resource. To get a notification when a previously defined threshold was reached the client can implement a call-back port.

```
// IDL
interface iResourceObserver {
    oneway void upperThresholdReached (
        in tThresholdID threshold,
        in tInstanceID resource,
        in tUsage currentUsage
    );
    oneway void lowerThresholdReached (
        in tThresholdID threshold,
        in tInstanceID resource,

```

```
        in tUsage currentUsage
    );
};
```

A.1.12 Interface iMonitoredResourceManager

The manager for monitored resources combines the functionality of the resource management and monitoring ports without defining any new operations.

```
// IDL
interface iMonitoredResourceManager :
    iResourceManager,
    iResourceMonitor
{ };
```

A.1.13 Interface iVirtualEnvironmentManager

A manager for virtual environments is a special resource manager and is used to create and destroy virtual environments (VEs). When a VE is created the associated resources have to be specified. This may include a demultiplexer channel to which service component instances inside the VE can be connected later. The VE manager has to ensure that the resources consumed by components assigned to the VE (via their owner) will not exceed the overall VE quota.

The VE manager offers an operation for retrieving a VE based on its virtual network identifier. This identifier is specified in the initial setup during the creation of a VE and is used to determine to which virtual network a VE belongs. Further, there is an operation for making other resource managers known to the VE manager.

```
// IDL
interface iVirtualEnvironmentManager :
    iMonitoredResourceManager
{
    tComponent getInstanceByVirtualNetworkID (in string ID);
    void attachManager (
        in tTemplateDescription description,
        in iComponentInitial initial
    );
    void detachManager (in tTemplateDescription manager);
};
```

A.2 IMPLEMENTATION OF PROMETHOS

PromethOS is implemented under Linux 2.4. The source code can be found on <ftp://ftp.kernel.org> or one of its mirrors. We apply the patches as required for kernel pre-emption. These patches can be found on <http://www.tech9.net/rml/linux>.

In the remainder of this section, we introduce the programming concepts as required for programming PromethOS plugins. We start with an introduction to the framework and highlight the issues that need to be addressed for programming a plugins. However, for sake of simplicity we restrict the exemplary explanation to IPv4.

A.2.1 PromethOS Netfilter-Table: iptables_promethos.c

The PromethOS table is defined in the file `linux/net/ipv4/netfilter/iptables_promethos.c`. Hereafter, we explain the details of this table.

PromethOS has been implemented as a Linux Kernel Module. Thus, the module provides the standard interface to loading Linux kernel modules, i.e. `module_init(init)` and `module_exit(fini)`, where `init` and `fini` identify the respective functions defined in the

PromethOS table-module. At initialisation-time of the kernel module, the function `init()` is called, while at removal-time, the function `fini()` is called. Thus, the following explanations refer to code implemented in the `init()` or `fini()` functions.

In `init()`, PromethOS needs to register its table at every Netfilter hook. This is required such that filters within the PromethOS table may evaluate packets at every hook-location of the path. The following statement in the file gives the definition of the hook-registration.

```
#define PROMETHOS_VALID_HOOKS \
    (1 << NF_IP_PRE_ROUTING) | \
    (1 << NF_IP_POST_ROUTING) | \
    (1 << NF_IP_LOCAL_IN) | \
    (1 << NF_IP_FORWARD) | \
    (1 << NF_IP_LOCAL_OUT)
```

In this statement, we define five hooks where the PromethOS table needs to register itself. The hooks themselves (`NF_IP_PRE_ROUTING`, `NF_IP_POST_ROUTING`, `NF_IP_LOCAL_IN`, `NF_IP_FORWARD`, `NF_IP_LOCAL_OUT`) are defined in the appropriate include-file of Netfilter.

The interface for the registration of the PromethOS Netfilter-table takes place by launching a predefined function call:

```
ipt_register_table(&packet_promethos);
```

This call should be self-explanatory. However, we explain the structure in more detail hereafter:

```
struct ipt_table packet_promethos = {
    { NULL, NULL },
    "promethos",
    &initial_table.repl,
    PROMETHOS_VALID_HOOKS,
    RW_LOCK_UNLOCKED,
    NULL };
```

This structure defines all information Netfilter requires for a new table. It defines the name of the table, the valid hooks – here; they are specified by `PROMETHOS_VALID_HOOKS` – as well as the initial state of the PromethOS table by specifying a pointer to the appropriate data structure (`&initial_table.repl`). `ipt_register_table()` is called in the `init()` function of the PromethOS.

Once the table has been registered, each hook needs to be specified. This specification (provided next) specifies the hooks, the hook priorities and the functions to be called. Besides that, the protocol needs to be specified as well.

```
struct nf_hook_ops ipt_opts[] = {
    {{ NULL, NULL }, ipt_hook, PF_INET, NF_IP_PRE_ROUTING,
    NF_IP_PRI_PROMETHOS },
    {{ NULL, NULL }, ipt_hook, PF_INET, NF_IP_POST_ROUTING,
    NF_IP_PRI_PROMETHOS },
    {{ NULL, NULL }, ipt_hook, PF_INET, NF_IP_FORWARD,
    NF_IP_PRI_PROMETHOS },
    {{ NULL, NULL }, ipt_hook, PF_INET, NF_IP_LOCAL_IN,
    NF_IP_PRI_PROMETHOS },
    {{ NULL, NULL }, ipt_hook, PF_INET, NF_IP_LOCAL_OUT,
    NF_IP_PRI_PROMETHOS } };
```

In this structure, `ipt_hook` specifies the function to be called, `PF_INET` defines the protocol family, `NF_IP_PRE_ROUTING`, `NF_IP_POST_ROUTING`, `NF_IP_FORWARD`, `NF_IP_LOCAL_IN` and `NF_IP_LOCAL_OUT` define the hooks and `NF_IP_PRI_PROMETHOS`

specifies the hook priority. In the `init()` function of PromethOS, this structure is used to register the function `ipt_hook()` at the appropriate hooks by launching the call to `nf_register_hook(&ipt_ops);`.

Arriving packets are processed in the function `ipt_hook()`. In this function, the packet is dispatched to the tables by launching the function call `ipt_do_table()`. This function sends the packet to the appropriate target. We quote the function from the source code, which is self-explanatory.

```
static unsigned int ipt_hook(unsigned int hook,
    struct sk_buff **pskb,
    const struct net_device *in,
    const struct net_device *out,
    int (*oknf)(struct sk_buff *))
{
    return ipt_do_table(pskb, hook, in, out, &packet_promethos,
        NULL);
}
```

In the `fini()` function, the hooks need to be unregistered; the following code fragment carries out this action:

```
for (I = 0; I < sizeof(ipt_ops)/sizeof(struct nf_hook_ops); i++)
    nf_unregister_table(&ipt_ops[i]);
    ipt_unregister_table(&packet_promethos);
```

We assume this code fragment would be self-explanatory.

A.2.2 PromethOS Netfilter-Target: `ipt_PROMETHOS.c`

The PromethOS target is implemented in file `linux/net/ipv4/netfilter/ipt_PROMETHOS.c`. Hereafter, we explain the important issues.

The PromethOS Netfilter-target is implemented as a Linux kernel module. So, the standard interfaces for modules must be followed. We do not explain them explicitly but focus on the implementation aspects.

PromethOS provides two macro definitions (`promethos_init()`, `promethos_exit()`), which are used by every PromethOS plugin to register itself. Internally, these macros are resolved to `promethos_register()` and `promethos_unregister()`. These functions are used for internal registration in the framework of PromethOS. They provide the following interface:

```
unsigned int promethos_register(char *promethosp,
    promethos_target_func_t t,
    promethos_config_func_t c,
    promethos_config_func_t rc,
    promethos_print_func_t p);

unsigned int promethos_unregister(char *promethosp);
```

The parameters are defined as follows:

- `promethosp` : provides the name of the plugin
- `t` : specifies the target function to be called for every packet
- `c` : identifies the configuration function that is called at initialisation time of a plugin
- `rc` : specifies the re-configuration function that is called when the user submits a re-configuration request by the `/proc`-filesystem interface.

- `p` : identifies the print function. This function is called when the user requests the plugin to report its current, gathered information via the `/proc`-filesystem.

For performance reasons, PromethOS dispatches packets to PromethOS plugins identified by the mechanisms of a hash functions (`ghash.h` of Linux, which usually needs some patches applied as distributed together with PromethOS). Two separate hash tables are implemented:

```
static struct target_table promethos_table;
static struct instance_table promethosi_table;
```

The corresponding structures are imported from the include file `ipt_PROMETHOS.h` which in turn uses the definitions provided by `ghash.h`. These two hash tables are used to locate the PromethOS plugins according to their name (`promethos_table`), or according to their instance number (`promethosi_table`) respectively. The instance number of a PromethOS plugin is known only after the call to the function `checkonfig()`, a function that controls for proper initialisation of the PromethOS plugins. This function is called at load-time of the PromethOS plugin. It launches the PromethOS-config function.

A.2.3 Plugin Implementation

A framework should ease the development of components running within a framework. Therefore, emphasize was put on the clear and simple interface specification for PromethOS plugins.

A PromethOS plugin is requested only to include the appropriate PromethOS header file, and to provide the following functions; they are explained in great detail afterwards:

- `load()`
- `unload()`
- `target()`
- `config()`
- `reconfig()`
- `print()`

A.2.3.1 load()

The load function of a plugin is provided such that additional initialisation may be carried out. This function is run exactly once during load-time of the PromethOS plugin. So, it is appropriate for plugin-global initialisation issues if required.

A.2.3.2 unload ()

The unload function is called during the removal of a PromethOS plugin. It may be used for the release of plugin-global resources for example.

A.2.3.3 target()

The target function is called for every packet of a flow. So it is called if a flow specification matches internally to the PromethOS table. To this function, the complete information is passed that accompanies a packet kernel internally. Thus, the function may evaluate or modify according to its own requirements.

A.2.3.4 config()

The config function is called for every plugin instance creation together with the respective configuration string. This configuration information is passed either by the `/proc`-filesystem

interface or by `iptables`.

A.2.3.5 `reconfig()`

The `reconfig` function is called every time a user process writes into the management interface provided by the `/proc`-filesystem. By this function, run-time re-configuration of a PromethOS plugin can be carried out. PromethOS passes the complete string of information as specified to the `/proc`-filesystem interface to the `reconfig` function. The `reconfig` function may be used further for reporting statistical information per instance.

A.2.3.6 `print()`

The `print` function is used to report data per plugin. This function is the only optional one, i.e. instead of this function; a predefined value may be specified to initialisation interface of PromethOS. This function can be used to report all information of a plugin by a single call.

A.2.4 Example Plugin Explained

Hereafter, we explain an exemplary implementation of a dummy plugin. However, we restrict the explanation to the points not mentioned previously.

```
#include <linux/netfilter_ipv4/promethos.h>
static int __init load()
{
    /* do some plugin initialisation stuff */
    return 0;
}

static void __exit unload()
{
    /* do plugin cleanup */
}

unsigned int promethos_dummy_target(struct sk_buff *pskb,
    unsigned int hooknum,
    const struct net_device *in,
    const struct net_device *out,
    unsigned long instance)
{
    /* process the packet */
    return IPT_CONTINUE;
}
```

The target function is called for every packet that the dispatcher sends to the plugin instance. It is the target function that decides how the Netfilter framework should proceed with the packet. The following return values are possible:

<code>IPT_CONTINUE</code>	Continue processing the packet along the chain of configured plugins
<code>NF_ACCEPT</code>	Accept the packet for further processing on the next layer
<code>NF_DROP</code>	Refuse the packet from further processing
<code>NF_STOLEN</code>	Declare the packet as plugin internally consumed
<code>NF_QUEUE</code>	Make Netfilter enqueue the packet for user space
<code>NF_REPEAT</code>	Repeat the actual hook.

```
unsigned int promethos_dummy_config(const char *config, unsigned long instance)
{
    /* do something */
    return 1;
}
```

The parameter `config` contains the string as specified by the user process. The instance identifies each instance of a plugin.

```
unsigned int promethos_dummy_reconfig(const char *config,
    unsigned long instance)
{
    /* do something */
    return 1;
}

unsigned int promethos_dummy_print(unsigned long instance,
    char *buffer,
    long bufferlength)
{
    /* fill buffer and return buffer length consumed */
    return 1;
}
```

The print function is called every time PromethOS requests the plugin to report internal state information. A pre-allocated buffer is passed to the plugin. The parameter `bufferlength` specifies the size of the buffer.

```
promethos_init("DUMMY",
  load,
  promethos_dummy_target,
  promethos_dummy_config,
  promethos_dummy_reconfig,
  promethos_dummy_print);
promethos_exit(unload);
```

The `promethos_init()` and `promethos_exit()` specify the interfaces as required for registration of a plugin. As introduced previously, they are both defined as macros in the header file.

A.2.5 The PromethOS User Space Library

The PromethOS User Space Library encapsulates the complexity for programming and configuring the PromethOS framework. It is used within FAIN to integrate the PromethOS framework with the VEM. In table Interface Definition 1, the full interface definition of the PromethOS User Space Library is provided.

```
#ifndef _PROMETHOS_LIB_H_
#define _PROMETHOS_LIB_H_

#include "promethos_lib_defs.h"

struct co_specification_s
{
  unsigned int no_out_ports; /* No. of output ports available */
  boolean ctrl_port_exported; /* TRUE if the component exports a ctrl intf. */
  char config[LOCAL_SIZE + 1]; /* initial configuration string as required for
                                PromethOS plugins */
};

typedef struct ve_descriptor_s ve_descriptor_t; /* VE : Virtual Environment */
typedef struct ee_descriptor_s ee_descriptor_t; /* EE : Execution Environment */
typedef struct co_descriptor_s co_descriptor_t; /* CO : Component */
typedef struct co_specification_s co_specification_t;

typedef struct demuxSpec_s demuxSpec_t; /* Additional specification of
                                         demultiplexing criteria if
                                         FAIN had any; -> not yet
                                         implemented! */

typedef struct resource_descriptor_s resource_descriptor_t;

#ifdef _PROMETHOS_LIB_C_
#include "promethos_lib_ve.h"
#include "promethos_lib_ee.h"
#include "promethos_lib_co.h"
#endif /* _PROMETHOS_LIB_C_ */

/* IP Hooks */
/* They are taken from include/linux/netfilter_ipv4.h */
/* They should be directly included from there!!!! */

/* After promisc drops, checksum checks. */
#define NF_IP_PRE_ROUTING 0
/* If the packet is destined for this box. */
#define NF_IP_LOCAL_IN 1
/* If the packet is destined for another interface. */
#define NF_IP_FORWARD 2
/* Packets coming from a local process. */
#define NF_IP_LOCAL_OUT 3
/* Packets about to hit the wire. */
#define NF_IP_POST_ROUTING 4
#define NF_IP_NUMHOOKS 5

#define ANYHOST -1U
```

```
#define ANYPORT -1U

typedef enum hook_descriptor_e
{
    PREROUTING = NF_IP_PRE_ROUTING,
    LOCAL_IN   = NF_IP_LOCAL_IN,
    FORWARD    = NF_IP_FORWARD,
    LOCAL_OUT  = NF_IP_LOCAL_OUT,
    POSTROUTING = NF_IP_POST_ROUTING
} hook_descriptor_t;

ve_descriptor_t *createVE(void);

ee_descriptor_t *createEE(ve_descriptor_t *VE);

co_descriptor_t *createCO(
    ee_descriptor_t *EE,          /* EE the new component should belong to */
    char *componentName, /* component to load, only plugin name, e.g WV
                             for the promethos_WV plugin */
    co_specification_t *co_spec); /* component specification */

pi_error_t deleteVE(ve_descriptor_t *VE);
pi_error_t deleteEE(ee_descriptor_t *EE);
pi_error_t deleteCO(co_descriptor_t *CO);

pi_error_t demuxCO(
    co_descriptor_t *CO, /* component for which the demux
                          * Condition must be met */

    char *inputInterface,
    char *sourcePort,
    char *sourceIP,
    char *outputInterface,
    char *destinationPort,
    char *destinationIP,
    char *protocol,
    demuxSpec_t *additionalDemux
);

pi_error_t demuxEE(
    ee_descriptor_t *EE, /* EE for which the demux
                          * Condition must be met */

    hook_descriptor_t hook, /* to which hook to attach to */
    char *inputInterface,
    char *sourcePort,
    char *sourceIP,
    char *outputInterface,
    char *destinationPort,
    char *destinationIP,
    char *protocol,
    demuxSpec_t *additionalDemux
);

pi_error_t addPort(
    co_descriptor_t *connect_to, /* component to connect to */
    co_descriptor_t *to_connect, /* component to connect */
    int portNo); /* output port No. of connect_to */

pi_error_t delPort(
    co_descriptor_t *co, /* component to acct on */
    int portNo); /* output port to free */

pi_error_t setmemLimits(ve_descriptor_t *VE, /* Memory */
    limit_t *limit_min, limit_t *limit_max);

pi_error_t setcpuLimits(ve_descriptor_t *VE, /* Processor Cycles */
    limit_t *limit_min, limit_t *limit_max);

pi_error_t setb_wLimits(ve_descriptor_t *VE, /* Bandwidth */
    limit_t *limit_min, limit_t *limit_max);

pi_error_t getmemLimits(ve_descriptor_t *VE, /* available Memory left */
    limit_t *limit);

pi_error_t getcpuLimits(ve_descriptor_t *VE, /* available Processor Cycles left */
    limit_t *limit);
```

```
pi_error_t getb_wLimits(ve_descriptor_t *VE, /* available Bandwidth left */
    limit_t *limit);

#endif /* _PROMETHOS_LIB_H_ */
```

Interface Definition 1: PromethOS User Space Library

The PromethOS User Space Library is compiled and linked into a standalone library. It can be used to connect an arbitrary application to the PromethOS framework provided that this language supports a C-linking style.

```
struct co_specification_s
{
    unsigned int no_out_ports; /* No. of output ports available */
    boolean ctrl_port_exported; /* TRUE if the component exports a ctrl intf. */
    char config[LOCAL_SIZE + 1]; /* initial configuration string as required for
        PromethOS plugins */
};
```

Every PromethOS plugin consists of a single input port and a specified number of output ports, which could be infinite theoretically. `Co_specification_s` represents the component specification. The values must be specified before a single component gets instantiated.

```
typedef struct ve_descriptor_s    ve_descriptor_t; /* VE : Virtual Environment */
typedef struct ee_descriptor_s    ee_descriptor_t; /* EE : Execution Environment */
typedef struct co_descriptor_s    co_descriptor_t; /* CO : Component */
typedef struct co_specification_s co_specification_t;

typedef struct demuxSpec_s demuxSpec_t; /* Additional specification of
        demultiplexing criteria if
        FAIN had any; -> not yet
        implemented! */

typedef struct resource_descriptor_s resource_descriptor_t;
```

The data structures defined above represent opaque data types to the user of the PromethOS User Space Library. Internally, they are used for the objects mentioned in the associated comments. These data structures are used as arguments and results mentioned in the methods described next. They have been implemented as opaque data types to the user such that only pre-defined interfaces may be used.

```
/* IP Hooks */
/* They are taken from include/linux/netfilter_ipv4.h */
/* They should be directly included from there!!!! */

/* After promisc drops, checksum checks. */
#define NF_IP_PRE_ROUTING 0
/* If the packet is destined for this box. */
#define NF_IP_LOCAL_IN 1
/* If the packet is destined for another interface. */
#define NF_IP_FORWARD 2
/* Packets coming from a local process. */
#define NF_IP_LOCAL_OUT 3
/* Packets about to hit the wire. */
#define NF_IP_POST_ROUTING 4
#define NF_IP_NUMHOOKS 5

#define ANYHOST -1U

#define ANYPORT -1U

typedef enum hook_descriptor_e
{
    PREROUTING = NF_IP_PRE_ROUTING,
    LOCAL_IN = NF_IP_LOCAL_IN,
    FORWARD = NF_IP_FORWARD,
    LOCAL_OUT = NF_IP_LOCAL_OUT,
```

```
POSTROUTING = NF_IP_POST_ROUTING
} hook_descriptor_t;
```

The constant values used in the PromethOS User Space Library interface definition are based on those defined in the netfilter framework. For simplicity reasons, they have been redefined in this interface definition file.

In the PromethOS User Space Library, of major importance is the data structure describing the hook (hook_descriptor_t) where the PromethOS EEs must be attached to. The specification of the hook according to hook_descriptor_t (PREROUTING, LOCAL_IN, FORWARD, LOCAL_OUT, POSTROUTING) corresponds to the hooks as shown in Figure 6-2. The hook_descriptor_t is used for the specification where a PromethOS EE should be hooked to in the method demuxEE() as explained below.

```
ve_descriptor_t *createVE(void);
```

A virtual environment (VE) is instantiated internally to PromethOS. The data structure representing the VE is returned in case of success. Otherwise, NULL is returned.

```
ee_descriptor_t *createEE(ve_descriptor_t *VE);
```

Within a VE, PromethOS EEs are instantiated by calling createEE. As an argument, the previously instantiated VE must be provided. On success, the EE-object is returned.

```
co_descriptor_t *createCO(
    ee_descriptor_t *EE,           /* EE the new component should belong to */
    char *componentName, /* component to load, only plugin name, e.g WV for
                               the promethos_WV plugin */
    co_specification_t *co_spec); /* component specification */
```

The createCO() method loads, if required, and instantiates the specified component (PromethOS plugin) within EE that is provided as an argument to the method. By the componentName argument, the PromethOS plugin to load is specified. Note that only the plugin name and not the file name must be specified. The component specification (co_spec) is used by this method for defining the facilities the PromethOS plugin provides. On success, the method returns the instantiated component, i.e. the object representing the instantiated component.

```
pi_error_t deleteVE(ve_descriptor_t *VE);
pi_error_t deleteEE(ee_descriptor_t *EE);
pi_error_t deleteCO(co_descriptor_t *CO);
```

By the methods deleteVE(), deleteEE() and deleteCO() the VE, the EE and the component (CO) as specified by the arguments are deleted and removed, if appropriate. Error codes are reported if removal fails. If objects still contain components or execution environments, they are removed by these methods.

```
pi_error_t demuxCO(
    co_descriptor_t *CO, /* component for which the demux
                          * Condition must be met */
    char *inputInterface,
    char *sourcePort,
    char *sourceIP,
    char *outputInterface,
    char *destinationPort,
    char *destinationIP,
    char *protocol,
    demuxSpec_t *additionalDemux
);
```

By the method demuxCO(), fine granular demultiplexing methods can be specified per component.

The method expects the previously instantiated component objects as the first argument. By the next seven arguments, the demultiplexing is specified according to the notion expected by the iptables tool. The argument `additionalDemux` is not used currently. Is defined for future, internal demultiplexing. For example the FAIN ANEP header-Flag could be specified there. Results are signaled by the return value.

```
pi_error_t demuxEE(
    ee_descriptor_t *EE, /* EE for which the demux
                        * Condition must be met */
    hook_descriptor_t hook, /* to which hook to attach to */
    char *inputInterface,
    char *sourcePort,
    char *sourceIP,
    char *outputInterface,
    char *destinationPort,
    char *destinationIP,
    char *protocol,
    demuxSpec_t *additionalDemux
);
```

By the method `demuxEE()`, the previously instantiated EE is first attached to a hook of the Netfilter framework, and the demultiplexing for data flows to be directed to that EE is specified. The arguments are identical to those explained for `demuxCO()` (see above).

```
pi_error_t addPort(
    co_descriptor_t *connect_to, /* component to connect to */
    co_descriptor_t *to_connect, /* component to connect */
    int portNo); /* output port No. of connect_to */

pi_error_t delPort(
    co_descriptor_t *co, /* component to acct on */
    int portNo); /* output port to free */
```

By the method `addPort()`, components are interconnected. The component specified by `to_connect` is connected to the one specified by `connect_to`. As a third argument, the port number (`portNo`) used by the output port is specified. Results are signaled by the return value. By the method `delPort()`, component interconnections are removed. The component from which the outbound connection to the next component is specified by `co`, the output port number to remove the connection from by `portNo`. Results are signaled by the return value of the method.

```
pi_error_t setmemLimits(ve_descriptor_t *VE, /* Memory */
    limit_t *limit_min, limit_t *limit_max);

pi_error_t setcpuLimits(ve_descriptor_t *VE, /* Processor Cycles */
    limit_t *limit_min, limit_t *limit_max);

pi_error_t setb_wLimits(ve_descriptor_t *VE, /* Bandwidth */
    limit_t *limit_min, limit_t *limit_max);

pi_error_t getmemLimits(ve_descriptor_t *VE, /* available Memory left */
    limit_t *limit);

pi_error_t getcpuLimits(ve_descriptor_t *VE, /* available Processor Cycles left */
    limit_t *limit);

pi_error_t getb_wLimits(ve_descriptor_t *VE, /* available Bandwidth left */
    limit_t *limit);
```

Per VE, the constraints can be set for consumption of memory, cpu-cycle and network-bandwidth by the set-methods. A credit system is used. The unused credits can be retrieved by the get-methods.

A.2.6 Example Use of PromethOS Plugin Framework

To give the reader a feel for the simplicity and elegance with which plugins can be put into operation, we illustrate the commands necessary to load and configure a WaveVideo [17] plugin performing video scaling. Note that these commands can be executed at any time, even when network traffic is transiting through the system. As mentioned above, we use a PromethOS-enhanced iptables program

that interacts with the iptables framework. In the extension of iptables, we implement calls to the insmod program, which serves as the primary tool to install Linux kernel modules.

- Loading and registering plugin:

```
# iptables -t promethos -A PREROUTING -p UDP -s 129.132.66.115\  
--dport 6060 -j PROMETHOS --plugin WV --autoinstance --config \  
65536
```

This command adds a filter specification to the PromethOS plugin framework, requesting to install the WV plugin at the PREROUTING hook, and creating an instance of this plugin to perform video scaling at 65536 Byte/s. If the plugin framework is not yet loaded, the module dependency resolution of Linux installs it on demand.

1. Upon successful completion of the plugin loading and instantiation, the plugin framework reports the plugin instance number:

```
PromethOS plugin instance is 1
```

2. By this instance number, the plugin control port can be accessed:

```
# echo '#1' 131072 > /proc/promethos/net/management
```

This reconfigures the WV plugin to scale the video to a maximum output of 131072 Byte/s.

3. The configuration of the PromethOS table can be retrieved with iptables:

```
# iptables -t promethos -L  
Chain PREROUTING (policy ACCEPT)  
target prot opt source destination  
PROMETHOS udp -- 129.132.66.115 anywhere udp dpt:6060 WV#1
```

4. The plugin and the framework may be removed from the kernel by the standard mechanisms provided by iptables and the Linux kernel module framework.

This example demonstrates the seamless integration of the PromethOS plugin framework in Linux, allowing to load arbitrary code at runtime.

A.2.7 Example Use of the PromethOS User Space Library

```
#include <unistd.h>  
#include <stdio.h>  
#include <stdlib.h>  
#include <string.h>  
#include "promethos_lib.h"  
int main(int argc, char **argv, char **envp)  
{  
    ve_descriptor_t *ve = NULL;  
    ee_descriptor_t *ee = NULL;  
    co_descriptor_t *co1 = NULL;  
    co_descriptor_t *co2 = NULL;  
    co_specification_t co_1_spec = {};  
    co_specification_t co_2_spec = {};  
    co_spec.no_out_ports = 5; /* five output ports are provided */  
    co_spec.ctrl_port_exported = FALSE; /* no export a control interface */  
    strcpy(co_spec.config, "1.2.3"); /* just a dummy configuration */  
    co_2_spec.no_out_ports = 1; /* only one output port is provided */  
    co_2_spec.ctrl_port_exported = TRUE; /* a control interface is exported */  
    strcpy(co_2_spec.config, "65536"); /* plugin specific configuration */  
  
    if ((ve = createVE()) != NULL) { /* create the VE */  
        if ((ee = createEE(ve)) != NULL) { /* create an EE within the VE */  
            demuxEE(ee, /* EE for which the demultiplexing should happen */
```

```
    PREROUTING,          /* attach to which hook */
    NULL,                /* source interface */
    NULL,                /* source port */
    NULL,                /* source IP */
    NULL,                /* destination interface */
    "26473",             /* destination port */
    NULL,                /* destination IP */
    "UDP",               /* protocol */
    NULL);               /* additional demux specification */

if ((col = createCO(ee, "LOG", &co_spec)) != NULL)
{
    printf("Installed and instantiated the PromethOS LOG plugin!\n");
    if ((co2 = createCO(ee, "WV", &co_2_spec)) != NULL)
    {
        printf("Installed and instantiated the PromethOS WV plugin!\n");
        if (addPort(col, co2, 1) == OK)
        {
            printf("Configured the two components such that col->co2!\n");
        }
    }
    else
        fprintf(stderr, "createCO(%p, \"test_2_component\", %p) failed\n",
            ee, &co2);
}
else
    fprintf(stderr, "createCO(%p, \"test_component\", %p) failed\n",
        ee, &col);
}
else
    fprintf(stderr, "createEE(%p) failed\n", ve);
}
else
    fprintf(stderr, "createVE() failed\n");

deleteEE(ee); /* remove the EE */
deleteVE(ve); /* remove the VE */

return 0;
}
```

Table 3: Example Code of the PromethOS User Space Library

In Table 3, an example use of the PromethOS user space library is provided. The comments available in the source code make together with the self-explaining method-names make the source code self-documenting.