Project Number :	**IST-1999-10561-FAIN**
Project Title :	**Future Active IP Networks**

D4-Revised Active Node Architecture and Design

CEC Deliverable Nr :	**D4**
Deliverable Type :	**CO (Delete as appropriate)**
Dissemination:	**Int**
Deliverable Nature :	**R**
Contractual date :	**April 2002 (as updated in Project Quarterly Reports)**
Actual date :	**Y2M12**

Editor :	**Spyros Denazis**
Workpackage(s) :	**WP3**
Abstract :	**2nd version of the FAIN Active Router Design**
Keyword List :	**Active Router, Virtual Environments, Execution Environments, Node Manager, Resource Control Framework, Demultiplexing, Security.**

Project Number :	**IST-1999-10561-FAIN**
Project Title :	**Future Active IP Networks**

D4-Revised Active Node Architecture and Design

Editor :	**Spyros Denazis**
Document No:	**D4**
File Name	**WP3-HEL-030-D4-Int.doc**
Contributors :	**WP3**
Version :	**1.5**
Date :	**Friday, 17 May 2002**
Distribution :	**WP3, WP4, WP5**

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)
Dusan Gabrijelcic (JSIS)
Thomas Becker (FHG/FOKUS)
Antonis Lazanakis (NTUA)
George Karetsos (NTUA)
Drissa Houatra (FT)
Lukas Ruf (ETH)
Alex Galis (UCL)
Walter Eaves (UCL)
Lawrence Cheng (UCL)
Kiminori Sugauchi (HIT)
Eckhard Moller (FHG/FOKUS)

Change History

Ver.	Date	Authors	Comments
0.0	04.10.2001	Spyros Denazis (HEL)	Baseline document of D2 and initial ToC
0.1	15.10.2001	Spyros Denazis (HEL)	Updated version due to feedback received
0.2	25.02.2002	Spyros Denazis (HEL)	New ToC and M3 Scenarios Initial draft
0.3	18.03.2002	Spyros Denazis (HEL) Toshiaki Suzuki (HEL) Dusan Gabrijelcic (JSIS) Thomas Becker (FHG/FOKUS) Antonis Lazanakis (NTUA) George Karetsos (NTUA) Drissa Houatra (FT) Lukas Ruf (ETH)	Initial input received for the following sections: Sections 3, 10.2, 10.3: WP3-FHG-031-D4-VEM.doc Sections 4, 10.1, 10.2, 10.3: WP3-HEL-041-D4-DeMux-Int.doc Sections 5, 10.1, 10.2, 10.3: WP3-JSIS-006-D4-rev_sec_arch_v01.doc Sections 6: WP3-NTU-XXX-RCF-IntV1.0.doc Sections 7.1, 10.2: WP3-ETH-001-D4-Int.doc
0.4	22.03.2002	Spyros Denazis (HEL) Toshiaki Suzuki (HEL) Antonis Lazanakis (NTUA) Lukas Ruf (ETH)	Editorial changes according to text format guidelines and additional input in sections: Sections 4, 10.1, 10.2, 10.3: WP3-HEL-041-D4-DeMux-Int.doc Sections 6: WP3-NTU-FT-004-RCF-Int.doc Sections 7.1, 10.2: WP3-ETH-001-D4-Int.doc
1.0	2.05.2002	Spyros Denazis (HEL) Thomas Becker (FHG/FOKUS) Toshiaki Suzuki (HEL) Antonis Lazanakis (NTUA) Walter Eaves (UCL) Lawrence Chang (UCL) Alex Galis (UCL)	Appendix on Scenarios and IDLs have been dropped and moved to R10 Updated version of VEM was added replacing the old one: WP3-FHG-041-D4-Int.doc New section, 7.1, was added describing Java EE: WP3-FHG-041-D4-Int.doc Updated version (V0.3) of Demux was added replacing the old one: WP3-HEL-041-D4-DeMux-Intv.doc Updated version (V0.6) on RCF was added replacing the old one: WP3-NTU-FT-007-RCF-Intv1.doc New section, 7.3, was added describing Active SNMP: WP3-UCL-004-R5-SNMPActivator.doc
1.1	9.05.2002	Spyros Denazis (HEL) Dusan Gabrijelcic (JSIS) Lukas Ruf (ETH) Thomas Becker (FHG/FOKUS) Alex Galis (UCL)	FHG name changed to FHG/FOKUS Revised section description of Component Manager Updated version of SEC was added replacing the old one: WP3-JSIS-003-D4-Int.doc Updated version of High Performance EE was added replacing the old one: WP3-ETH-004-D4-Int.doc

			FAIN Overview Section was added: WP3-UCL-040-D4.V2.doc Section 2, about Revised AN node architecture was added. Security section about architectural changes since D2 was brought forward to section 2.
1.2	13.05.2002	Spyros Denazis (HEL) Walter Eaves (UCL) Lawrence Chang (UCL) Alex Galis (UCL) Dusan Gabrijelcic (JSIS)	New Section added by Spyros Denazis (HEL) called "FAIN Architectural Ingredients". Revised version of Active SNMP section Editorial Comments submitted by Alex incorporated Editorial Comments submitted by Dusan (JSIS) incorporated
1.3	14.05.2002	Spyros Denazis (HEL) Kiminori Sugauchi (HIT)	Executive summary completed Conclusions section completed Last Version (0.3) of Components table was added as Appendix B
1.4	15.05.2002	Spyros Denazis (HEL) Alex Galis (UCL)	Editorial Comments submitted by Alex Galis incorporated. Other minor editorial changes
2.0	17.05.2002	Spyros Denazis (HEL) Eckhard Moller (FHG/FOKUS)	Editorial Comments submitted by Eckhard Moller (FHG/FOKUS) incorporated. The major change carried out as result of these comments was a revised introduction of the section about EEs to connect with the section about architectural ingredients and make the distinction between types and instances of EEs clear. Other minor editorial changes. Final Version of the Deliverable.

Executive Summary

D4 is the second in a series of three deliverables (i.e. D2, D4, D7) that are part of WP3 activities. It builds on D2, which was the first deliverable in the series and described the initial FAIN Active Network (AN) node architecture. An early version of the deliverable D4 was issued in early March 2002 as R5 & R6 internal reports. It describes both the revised FAIN active node architectural layers (i.e. R5 report) and the revised FAIN active node design (i.e. R6 report). As a follow up of D2, we have decided not to repeat parts of D2 here, but rather to focus on those parts the have undergone considerable revisions as a result of the implementation efforts and growing experience with the subject in year 2.These revisions may be classified into two categories: a) Concept revisions and b) Technical Revisions.

Concept revisions refer to the main architectural concepts outlined in D2 in that they needed more focus in some cases or lacked completeness in the previous version of the deliverable. In this deliverable we have revisited them 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 Node 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.

More specifically, section 1, named FAIN Overview, provides an overview of the FAIN active node and network architecture that includes results from other work packages and assists the reader of the deliverable in understanding the general picture of FAIN.

Section 2, named, FAIN Active Router Architectural Ingredients, summarises the concept revisions which was the result of the project's intention to aim at designing a system that is flexible and interoperable since Active Network technology is particularly suited for systems with these properties.

Section 3, named Revised Active Node Architecture, provides an overview of the revised node architecture focusing on Node OS and node management aspects. It also provides a summary for each one of the major components that appear in the architecture highlighting the differences since the previous deliverable, D2.

Sections 4-7, provide a detailed description of each one of the FAIN Active Router components, namely, VE Manager, Demux, Security and RCF, respectively, that includes their design and engineering aspects of their implementation.

Section 8, named, Execution Environments, describes three different execution environments that were the direct result of the realisations of the FAIN concepts outlined in section 2. Two of them, the JAVA EE and the High Performance EE, named PromethOS, have been designed and implemented as a proof of concept for the kind of flexibility that we envisage in FAIN. They build on the concept of component-based systems whereby services are downloaded in the form of components and are placed in the data path layer in order to process packets. The third EE, described in section 8.3, named SNMP Activator, is an example of a control EE from which the node interface (in this case it is an SNMP based interface) may be accessed. Such EE may be used by signalling protocols and it builds upon the SNAP EE, the background work of one of the FAIN partners, UPEN.

Finally, in Appendix A of the document, we have extended and enhanced the ANEP protocol to suit for the needs of FAIN for our testbed trials. In Appendix B, we provide an overview of the current status of the work in WP3 by means of distinct node components with their functionality.

D4 also represents the theoretical aspects of the architecture. The actual implementation that was built for M3 milestone is described in the R10 report, which also includes the scenarios to be demonstrated and the interface specifications, also a result of this project.

Comparing the two deliverables, D2 puts the emphasis on the design aspects of FAIN while D4 on the implementation aspects. Accordingly, the two documents may be viewed as complementary to each other and collectively they represent the current state of project achievement. Our intention is to merge the two documents into the third deliverable which will then represent the final result of WP3 including of course any changes and modifications that will occur as a result of the FAIN testbed deployment and evaluation in Year 3.

Table of Contents

Table of Figures

Table of Tables

1 FAIN OVERVIEW

1.1 Active Networking Issues in FAIN

The FAIN active network architecture defines active nodes, which provide full flexibility to the user for network management and service provisioning. The defining characteristic of an active node is the ability for users to load and manage software components dynamically and efficiently. This can be achieved safely since customers who are sharing the same active node would be provided with a VPN-like resource partitioning.

Packets requiring active processing are marked to allow correct handling by active routers. This allows the discrimination of active and conventional packets and the selection of an active node. Routing and node resources configuration in the active nodes could be achieved by setting policies at the network management level (element and network management nodes). Access to this functionality will be controlled and only possible via a well-defined API.

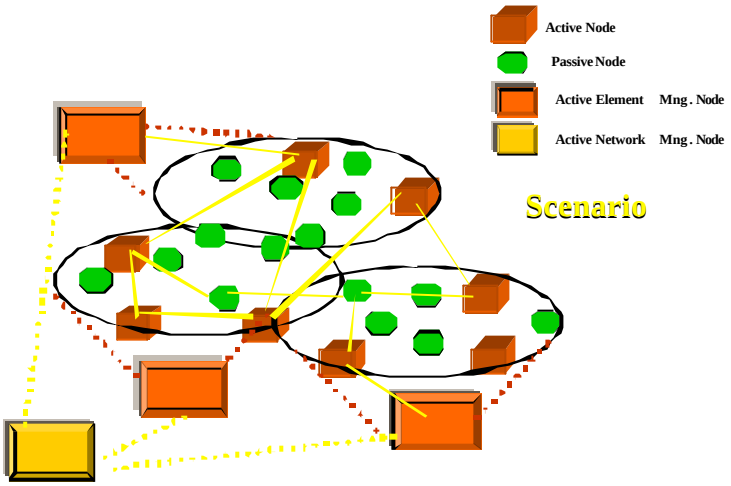


Figure 1-1: Management of Active Networks in FAIN

Figure 1-1 exemplifies a configuration of an active network and its management nodes.

1.2 Components in the FAIN Active Node

In relation to D2, we provide a summary of issues that are pertinent to WP3. The FAIN Reference Architecture consists mainly of the following entities: AA, EE and Node OS.

- *Active Applications/Services* (AA) are applications executed in active nodes.
- *Execution Environments* (EE) are environments where application code is executed. A privileged EE manages and controls the active node and it provides the environment where network policies are executed. Multiple and different types of EE are envisaged in FAIN. EEs are classified into virtual environments (VEs), where services can be found and interact with each other. VEs are interconnected to form a truly virtual network.
- *NodeOS* is an operating system for active nodes and includes facilities for setting up and management of communications channels for inter–EEs and AA/EEs, manages the router resources, provides APIs for AA/EEs, and isolates EEs from each other. Through its extensions the NodeOS offers facilities through the following components:
    - *Resource Control Facilities* (RCF). Through resource control, resource partitioning is provided and VEs are guaranteed that consumption stays within the agreed contract during an admission control phase, whether static or dynamic.

- *Security Facilities* (SF). The main security aspects are authentication and authorisation to control the use of resources and other objects of the node such as interfaces and directories. Security is enforced according to the policy profile of each VE.
- *Application/Service code deployment facilities* (ASP support). As flexibility is one of the requirements for programmable networks, partly realised as static or dynamic service provisioning, the NodeOS must support code deployment.
- *Demultiplexing facilities* (DEMUX). As flows of packets arrive at the node, Demultiplexing filters, classifies and diverts active packets to the appropriate VE, and consequently to the destination service inside the VE.
- *Node Management Facilities* (NM). The main aspects are the initiation and maintenance of VEs, control and management of the RCF and SF, management of the mapping of external to node policies into node resource and security configurations.

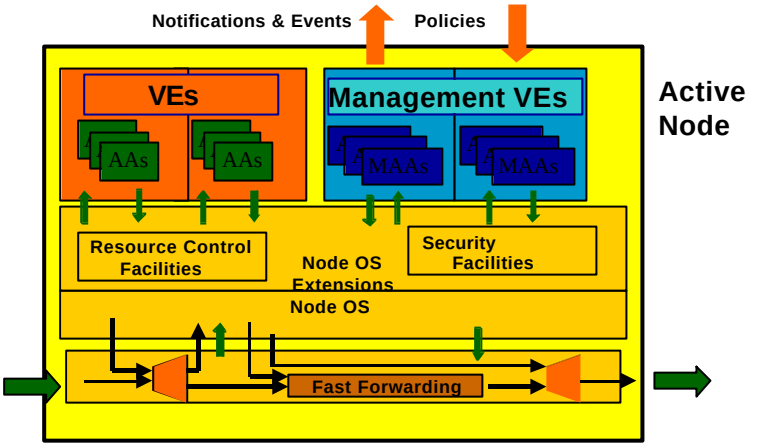


Figure 1-2: FAIN Active Node

Figure 1-2 describes the main design features and the components of the FAIN nodes:

In FAIN, node prototypes that are under development include: a high performance active node, with a target of 150 Mb/s; and a range of flexible and very functional active nodes/servers, with the target on multiple VEs hosting difference EEs

The common part of the prototypes (the FAIN middleware) is the NodeOS with the relevant extensions. Further details and discussions about the active node are provided in the remaining of this deliverable and in D2.

1.3 FAIN Active Management Components

Deliverable D5 will elaborate on the management approach in the FAIN project, which takes policy-based approach.

We envisage that the management of the active network will require the following features:

- *Policies:* Description of policies required to manage the active nodes and network
- *Node management component:* Design of management components within the active nodes, which will execute policies within an active node and monitor the local node resource usage. The execution of policies means mapping target policies into node resource configurations
- *Management stations:* A set of management nodes that will provide mechanisms to enable network administrators to manage the active networks as a whole, including network policies set-up and processing.

As the delivery of services will require co-operation of a number of active nodes the network providers will need the means of managing the active nodes as a group of nodes and not individual

nodes. They will need monitoring mechanisms for checking that correct policies are being defined and used in relation to the network before they are sent to the actual network. It will need to know what policies are currently loaded in the active nodes and what impact these are having on the network. It will also need to protect and monitor the security of the network. Therefore, the network/service provider needs a set of management mechanisms that will enable it to manage the network as a whole.

In FAIN we see the need for two types of management nodes in order to provide these mechanisms:

- Element Management Stations (EMS)
- Network Management Stations (NMS)

The main difference in functionality provided by these two types of management nodes is in the policy types, which they could process and manage, in the sub-networks, which they cover and in the creation of management domains for different types of users, as shown in Figure 1-3.

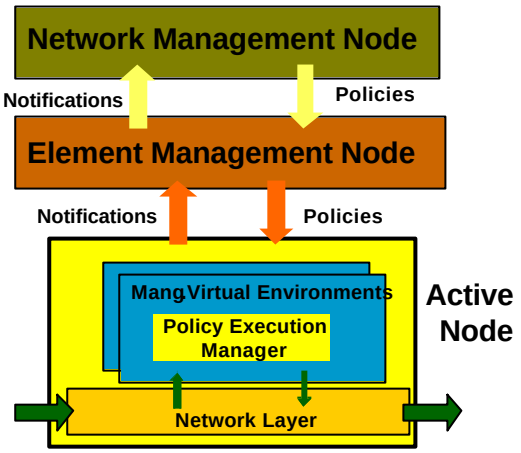


Figure 1-3: Active Network Management

Furthermore, the relationships between the EMS, NMS, and active nodes with regards to the policy flow are shown in Figure 1-3.

2 FAIN ACTIVE ROUTER ARCHITECTURAL INGREDIENTS

As a result of the analysis carried out in the previous deliverable, D2, we have focused on the following architectural ingredients as the most critical for realising the FAIN Active Router architecture:

- The Execution Environment(s)
- The Open Interfaces
- The Network Element

We have analysed and used these architectural ingredients with a clear separation between Control/Management and Transport planes and with the introduction of the interoperability plane.

The questions we faced were: a) how do we realise each one of them, and, b) how can they be combined (integrated) to fulfil the project objectives? Answering these questions gave rise to the details of the AN node architecture (see section 3 onwards) and provided a proper justification for the validity of the FAIN reference model (the "cube") argued in D2. In the remaining of this section we provide a description of these architectural ingredients highlighting on the design objectives thereof manifested as specific properties.

2.1 The FAIN Execution Environments

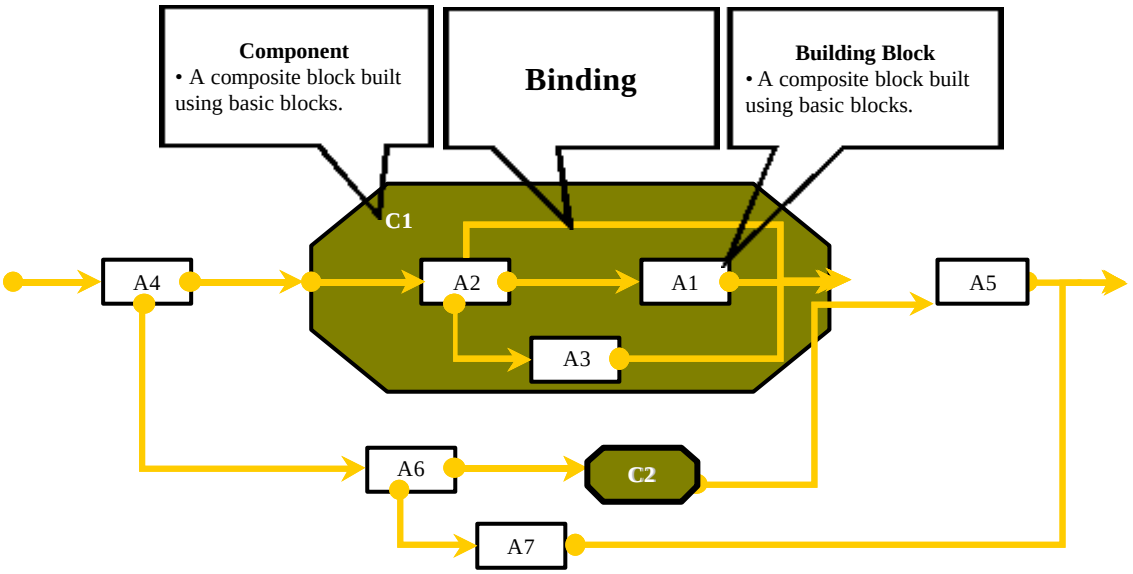


Figure 2-1: The Programming Environment

There have been a lot of views on the definition of an Execution Environment (EE) among researchers. Elaborating further on the analysis outlined in D2, and drawing from an analogy based on the concepts of *class* and *object* in object-oriented systems, we have distinguished between the *type* of an EE and the corresponding *instances* thereof.

An EE type is characterised by the programming methodology and the programming environment that is created as a result of the methodology used. In contrast, an EE instance represents the realisation of the EE type in the form of a runtime environment by using specific implementation technology e.g. programming language and binding mechanisms to maintain operation of the runtime environment. Accordingly, any particular EE type may have multiple instances while each instance may rely on different implementations.

Such distinction allowed us to address the issue of the principles that must govern and the properties that must be possessed by systems in next generation networks, from the issue of how to build such systems.

In FAIN one of the properties that has been the centre focus of our design and implementation efforts is *flexibility* and in particular, *composability* and *extensibility* aspects thereof.

The programming methodology that was used as part of the FAIN EE type was the building block approach according to which services break down into primitive, distinct blocks of functionality, which then may be bound together in meaningful constructs. To this end, services can be rebuilt from these primitive forms of functionality, called building blocks, while the building blocks may be reused and combined together in series of different arrangements as this is dictated by the service itself. The result of this process is the creation of a programming environment like the one depicted in Figure 2-1.

In FAIN we have built two different EE instances, namely, Java EE and High Performance EE, of this type described in section 8.

2.2 The Open Interfaces

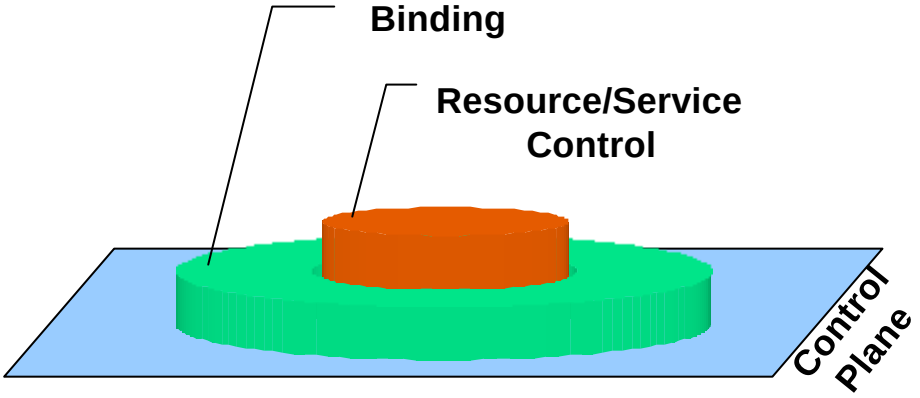


Figure 2-2: The building block abstraction

The next ingredient to our FAIN Active Router architecture is the identification and specification of open interfaces. While their scope varies from node level to network level, we primarily focus on the node level as this is the level where high level service requirements will be translated (mapped) into low (node) level interface accesses for the purposes of configuration and control of the node functionality thereby resulting in the behaviour expected by high level services.

This is also the level where standards are required so that services at different levels may access the same specification across different implementations. However, taking into account the fact that the node must be able to support flexibility a standard specification must adhere to the following principle:

> *Standardisation should not interfere with the development of new technologies and become a confinement to technological advancements but be capable of accommodating them.*

From the outset of the principle above, it seems that there is a conflict between the programming methodology which aims at creating component-based programming environments with the property of extending and reconfiguring themselves and a standard which implies that the functionality exported by the set of interfaces is fixed and only changes at a very slow pace. In other words, how is it possible for a standard to evolve at a similar pace with the technological advances? The answer can be found in the observation that such specification must also possess the same property, namely, flexibility and in particular composability and extensibility.

In FAIN we have captured the semantics of the composition mechanism as part of the node open interface. To this end, the control part of any new functionality is enhanced by a set of generic classes that offer methods that allow any control client to deploy into the node this new node functionality while binding it with other components at runtime. Figure 2-2 depicts how the enhanced control interface of functionality encapsulated in the form of a building block looks like from the viewpoint of the control plane. Such interfaces that participate in composability operations have been specified as part of the VE management framework described in section 4.

2.3 The Network Element

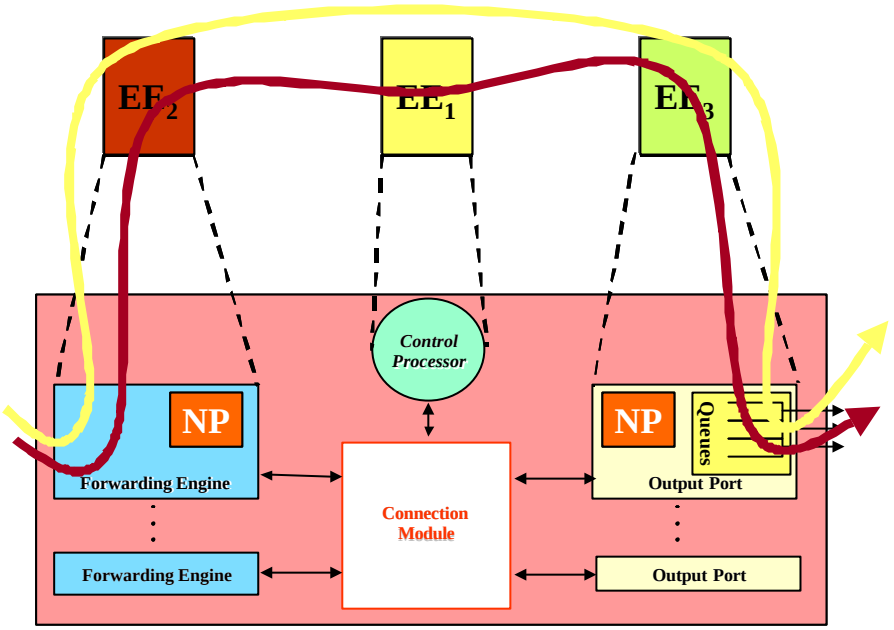


Figure 2-3: The Network Element Model

The previous two ingredients, namely the EE instances and the Open interfaces require a network element to reside in. Packets arriving at the node have to follow different data-paths inside the node. At every part of the node EEs have been instantiated implementing the same programming methodology and creating component-based programming environments. This gives rise to a new generation of network elements with architectures that are component-based. Such trend has been accelerated by the advent of innovative network products like the Network Processors (NP). Figure 2-3 depicts the new environment in the form of a Network Element Model.

In FAIN we have designed and built a prototype of an AN node that approximates the scenario above. Instead of a NP we have built one EE at the kernel space and another one at the user space. Both EEs support composability and receive packets, which are then directed to specific components as part of their data-path node crossing. More detailed description is provided in R10 as part of M3.

2.4 The Interoperability Layer

One goal of the FAIN Active Router architecture is the realisation or support of interoperability while preserving backward compatibility.

It should be noted that EEs are not being restricted to the transport plane and they apply also to the control & management planes. Since the control/management planes control the network element (which in turn controls the data-path creation), the control/management EEs can control the data-path behaviour. As different control/management EEs can implement the same open interface specification, this open interface specification can then abstract the structure of different data-paths. The control and management interfaces are therefore exported inside different control/management EEs. It is this open interface specification that defines the interoperability layer.

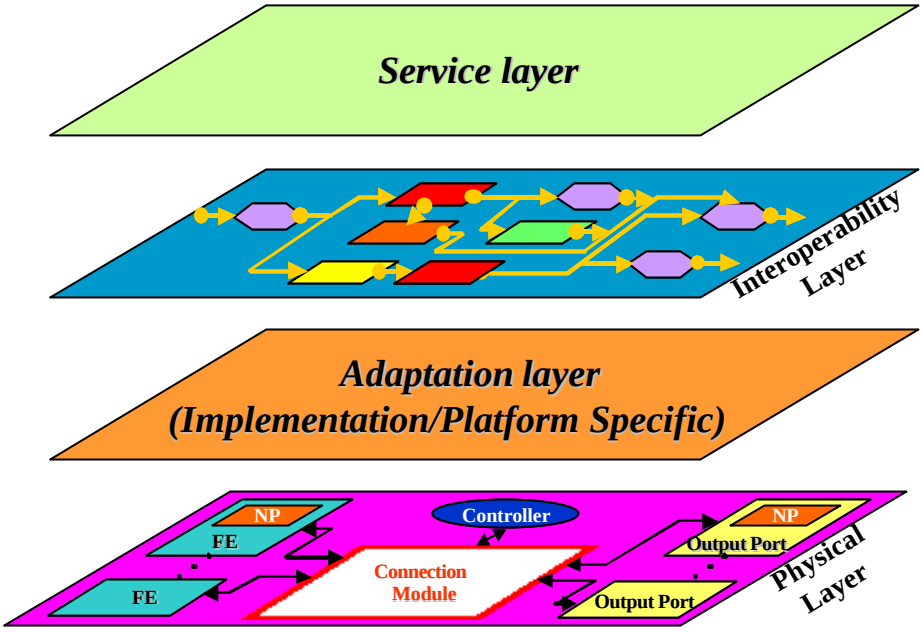


Figure 2-4: The Interoperability Layer

One of the major results of the FAIN approach is the creation of a layer that allows services to be deployed in heterogeneous environments. Recall from D2 that the FAIN reference model allows EEs to coexist within the same node in a synergistic way using each other's capabilities and services. Accordingly, EEs that exist in the transport plane and support specific component-based data-paths can be controlled by EEs in the control and management planes.

Creating an interoperability layer requires that the same open node interface specification - including the binding part used for composition purposes - must be implemented by every EE (EE instance) in the transport plane. In addition, in every C/M EE instance, control interfaces that adhere to the same specification must be provided in an EE specific format and connected with those in the transport plane that represent the actual implementation. The connection is the subject of what we call adaptation layer and its equivalent is the driver's layer of an operating system (Figure 2-4).

The specification itself represents the interoperability layers, which is realised inside the different EE instances of the control and management planes (Figure 2-4). In FAIN we have demonstrated this during the Opensig2002 demo and it is described in R10 of M3. In section 8.3, we describe a control EE that is built on the principle of control and transport plane separation and access an SNMP based interface that is exported inside the control EE instance (SNAP EE).

3 REVISED ACTIVE NODE ARCHITECTURE

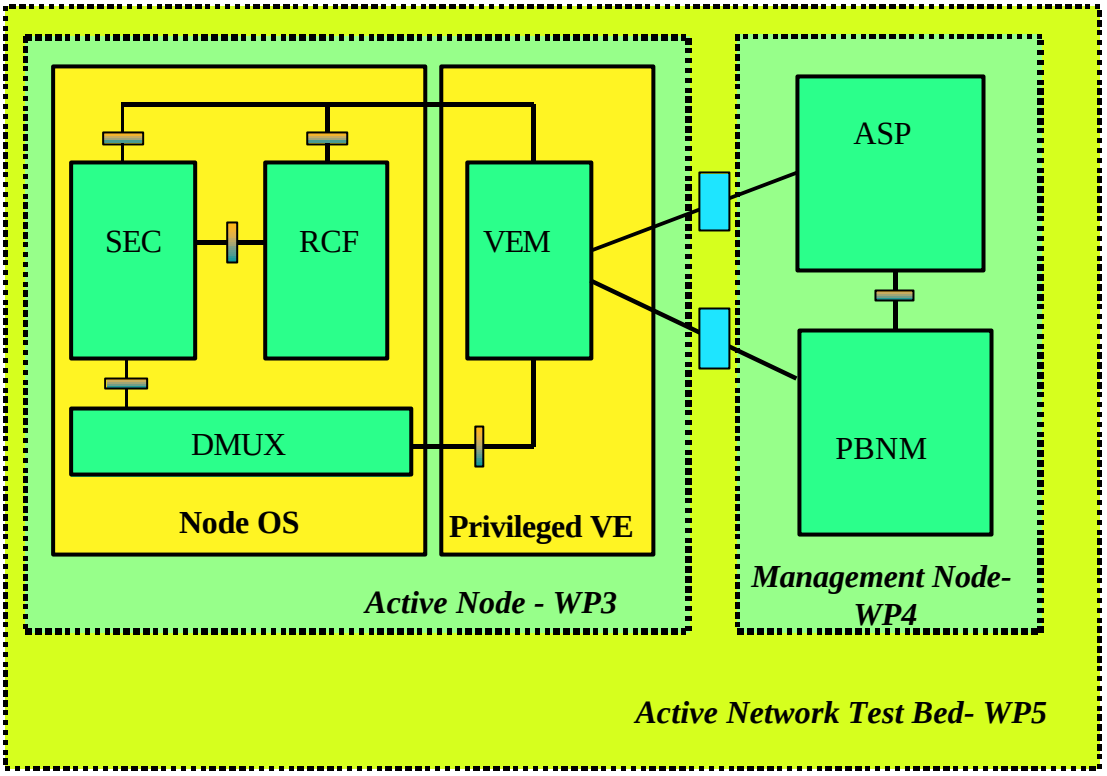


Figure 3-1: Overview of AN node architecture

In the section 2, we have highlighted some architectural ingredients and argued on the principles and objectives in using them in designing the FAIN Active Router architecture and its corresponding implementation. In this and subsequent sections we describe an architecture that integrates among the different components and how the ingredients have been used.

The AN node architecture has continued to evolve and in its revised version it reflects changes that were carried out as a result of the implementation and integration efforts. The changes pertain to the detail design of each one of the major node components and the specification of the inter-component interfaces. Figure 3-1 depicts the AN node overview of the major components and their corresponding interfaces that comprise the FAIN AN node architecture. Note that the figure shows the logical, not necessarily the physical distribution of the main components or their subparts. In this section we provide a summary and comment on the changes of the original AN node architecture, which is still valid and may be found in D2. Detailed description of the architecture and the changes are given in the corresponding sections that describe the components.

More specifically, The Privileged Virtual Environment has been enhanced with a new component, called VE manager, which implements the VE management framework. This component is the most crucial one as it offers a number of node services that are deemed necessary to configure and set-up the node. It is used for instantiating new VEs, deploying EEs and components therein as well as control interfaces that allows services inside VEs to customise resources according to application-specific requirements. In addition, the proposed framework allowed the implementation of other components like resource managers in RCF or channel managers in DMUX to be easily integrated with the implementation of the framework by means of a set of classes from where these components inherited. Finally, the VEM manager specified another set of interfaces, namely the Template Manager and Component manager that facilitate communication and integration with types of EEs other than the one that the VEM used for its own implementation. This will enable future integration with other implementation instances that are currently under development.

The DMUX has been enhanced with a Channel manager that is capable of creating different types of Channels, which are used to receive different types of packets, e.g. data packets or ANEP packets, and consequently deliver these packets to the proper services that are running inside different VEs. These channels are created by the DMUX components and are given to the requested VEs, which control them. To this end, VEs may request the creation or deletion of channels as well as configure these channels to receive certain packets, e.g. a specific IP address. The DMUX actually provides the two-ends (input and output) of a plug and play data-path that is supported by the component-based AN node architecture.

The goals of security architecture as stated in D2 remain the same. The architecture itself was further developed and is now more precisely defined. Two new entities were added, namely, the Security Manager and the Connection Manager. The former impacted the authorisation functionality, which now supports multiple authorisation engines, and exports security interfaces (policy and credentials) to the node. The latter, provides security support for hop-by-hop data integrity over connections with adjacent nodes. To this end, the security related options of the ANEP header were also specified and used in scenarios that involve this aspect of security functionality.

The AN node resource control (RCF) has adopted the VEM framework whereby the resources and their corresponding resource managers of the original RCF architecture in D2 are encapsulated in the Components manager of the VEM. Accordingly, the RMs can be deployed and controlled as any other regular service component through the interfaces inherited from the Component Manager. The RCF has also been extended with an Admission Control entity that is responsible for deciding whether the new VE creation request may be accepted provided that there are resources available left in the AN node.

The revised AN node architecture has been implemented in a Java EE on top of a Linux operating system using the Netfilter for packet classification and forwarding. This implies that the management functionality of the node has also used a Java EE. However, another aspect of the architecture is the simultaneous support of multiple EEs provided by the VEM framework. To this end, we have built two different types of EEs: one high performance EE in the kernel space of the Linux operating system and one control EE, called Active SNMP, which is based on SNAP EE.

The high performance EE is capable of dynamically deploying service components in the data-path on behalf of different VEs. The Active SNMP EE is an example of in-band signalling EE that enables valid users to control node resources by communicating with an SNMP agent.

4 VIRTUAL ENVIRONMENTS & MANAGEMENT

The FAIN active node allows for a variety of execution environments tailored to the specific needs potential active services may have. However, in order to manage different services in different environments a dedicated management environment is needed. The role of the management environment is to provide a uniform interface for accessing services running in the various execution environments. It supports interfaces for the installation, creation, and configuration of service components, especially the initial set-up of connections between components. The communication between components during their runtime can happen in whatever way is most effective and does not necessarily involve the management environment.

The concept of virtual environments is used to associate service components and their resource consumptions with the appropriate identity. We envisage one dedicated virtual environment owned by the node provider, however there could be several virtual environments owned by several service providers. The mapping from virtual environments to execution environments is not necessarily one-to-one, meaning that components represented in a virtual environment can be spread over different execution environments and components residing in an execution environment can be represented in different virtual environments.

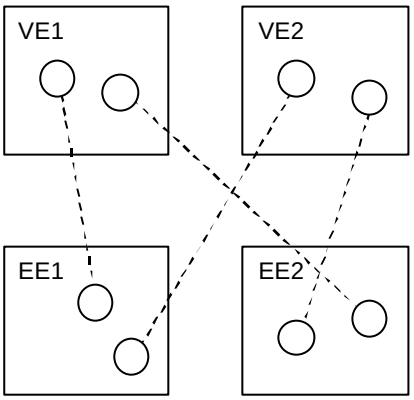


Figure 4-1: Mapping of components viewed on the virtual and execution level.

The collection of APIs described in this document provides the interface between the network management and the active node. The active service provisioning (ASP) part will use the *template manager* of the node provider to install new service types on the node. Subsequent services can be instantiated with the help of the *component manager*. A variety of *resource managers* are then used to allocate and monitor resources. The policy decision and enforcement points known from the policy based network management can be realised as components inside the management framework. In this way they can easily interact with already installed components – like resource managers and monitors – and make their own functionality available to other components.

After presenting the overall design - the individual parts will be outlined in more detail. Finally a collection of scenarios will illustrate the usage of the APIs.

4.1 Design

The three basic types of interfaces residing in the management environment are *iTemplateManager*, *iComponentManager*, and *iComponentInitial*. The management environment allows adding new interfaces to components so that they can offer their specific functionality.

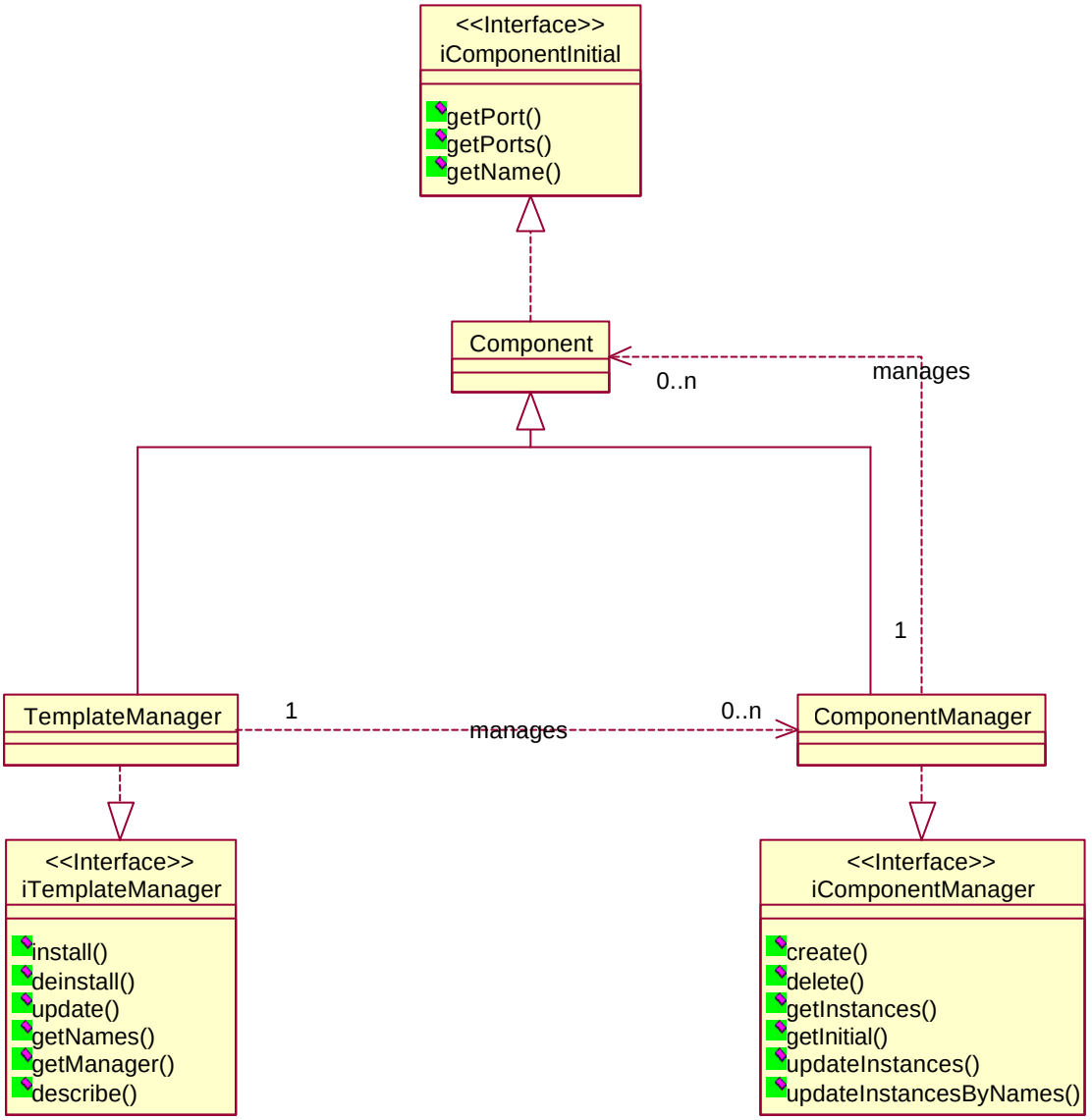


Figure 4-2: UML description of the VEM framework interfaces.

The interface *iTemplateManager* is assigned to a particular execution environment or virtual environment. It offers operations for installing templates in the managed environment, which means making code available for the creation of instances of a particular component type. Installation of a template includes the set-up of a manager for component instances; the manager's interface *iComponentManager* is the return value of the installation process. In the case that a template manager is assigned to a virtual environment all requests have to be forwarded to the manager of the appropriate execution environment because this is the place where the instances of component live.

The interface *iComponentManager* offers operations for managing instances of components including creation, listing, and deletion. The parameters for the creation of a component are specific to the type managed by the component manager (as defined by the respective template) and the result is the component's *iComponentInitial* interface. The component manager uses the parameters to initialise the new component instance. Where supported the parameters may also contain a description of the resource bounds for the new component. In this case the component manager acts as a resource manager and allocates the initial resources while it monitors the component's resource consumption

during runtime. Note that not all component managers need to support the features of a resource manager.

The interface *iComponentInitial* is used for the initial access of a client to a component. In order to get access to the component the client has to authenticate itself at this interface. The parameters for the authentication are the credentials of the client and the result is the component's requested specific interface tailored for the calling client.

The abstraction of *ports* is used to represent the connections a component provides to the outside world where interfaces as mentioned above are just a special kind of ports. Ports can be used for information exchange in any format or they can be used for expressing dependencies among component instances. Using the interface *iComponentInitial* a client can retrieve the references to the other interfaces and ports supported by the component.

4.2 Components

All components have a type, a unique name and an owner. The configuration of a component is described by properties. Interested clients can register for getting notified when particular properties change. To get access to the component's API a client has to authenticate itself first. The component manager that created the component sets the name and the initial configuration. Components can offer multiple interfaces in addition to the basic ones, e.g. a bandwidth manager will offer an interface for reserving bandwidth in addition to the interface for configuration.

4.2.1 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 has a name valid in the context of the holding component as well as a reference to the component itself. A port is described by a direction, 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 an optional type used for typed ports (e.g. an interface repository ID).

4.2.2 Properties

Instances of components are identified by a unique name and are owned by an identity. An identity is defined by its name and credentials. Assigning an owner to each component instance allows to control access and to do accounting both based on identities. Additionally components can have a number of freely definable properties. Such a property is constituted by its name and a value expressed as a CORBA Any. Properties can also be used to define a component's behaviour, e.g. a property may define a resource limit for a particular user or may restrict the access and usage of a component's interface.

4.2.3 Configuration

A component exposes an interface for configuration issues. At this interface it is possible to get and set properties as well as to register for property observation. An observer can implement call-back operations to get notifications about specific properties or about the set of properties as a whole. The call-back interface can be registered at the respective component.

The name and owner of the component can also be retrieved at the configuration interface. Additionally the ports of the component can be set-up and connected to other ports.

A component can be started and stopped. A newly created component is in stopped state. Starting and stopping together with getting and setting the component's properties is particularly useful to achieve mobility: stop the component, get the component's properties, re-instantiate the component at a different location, set the previously retrieved properties, start the new component.

4.3 Component Manager

A component manager is used to manage instances of components associated to a specific template, thus it is acting as a component factory. Managing comprises the creation, activation, deactivation, discovery, deletion, and update of instances. Creating a component instance is done by setting up a placeholder for the instance and use the specified resource profile to check the availability of the required resources. This may require to contact other managers for creating according resources. The ID of the new instance will be determined by the component manager and has to be unique; the owner will be set to the identity of the caller as detected during authentication. There is also an operation to explicitly set the owner during creation.

After a component instance was successfully created - i.e. all required resources are available - the instance may be activated specifying an initial set of properties. Activation will put the instance into a running state and make it functional. Now it is possible to retrieve the component's initial interface and start interaction. During runtime a component instance may be suspended and resumed. The actual implementation of this is component specific. Instances may also be deactivated and eventually deleted. When deactivating an instance it is possible to get the final state as a set of properties. This is useful for replacing an instance with a new one.

The component manager can also support updating of instances when a new template was installed. During an update a component will be deactivated and the state as represented by the component's properties will be saved. Then a new instance will be created from the recent template and activated with the previously saved properties.

All component managers offer an operation to retrieve the list of current component instances. Specific managers may additionally offer operations to find instances by specific features.

4.4 Template Manager

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 always includes a factory for the instances, which will be created from the template. This factory is called a component manager.

A template manager is assigned to a virtual environment (VE) representing a specific service provider or to an execution environment (EE). A special template manager is assigned to the privileged virtual environment representing the node provider. Figure 4-3 below depicts a possible layout of managers for two virtual environments and three execution environments. The EE_2 is of a type from which only one instance can exist (e.g. kernel or hardware) and is shared among VEs, EE_1 and EE_2 may be different instances of the same type (e.g. JAVA virtual machines).

The active service provisioning (ASP) module will use the node manager to install a new template on the node. This is done by passing a description of the template to the template manager of the node provider, which will forward the request to the template manager of the matching VE, which in turn will forward it to the template manager of the matching EE.

A template description comprises:

- the name of the template,
- the version of the template,
- the name of the class implementing the component manager,
- the path to the archive of the template,
- the VE identifier,

- the EE type identifier, and
- optional additional properties defining template specific features.

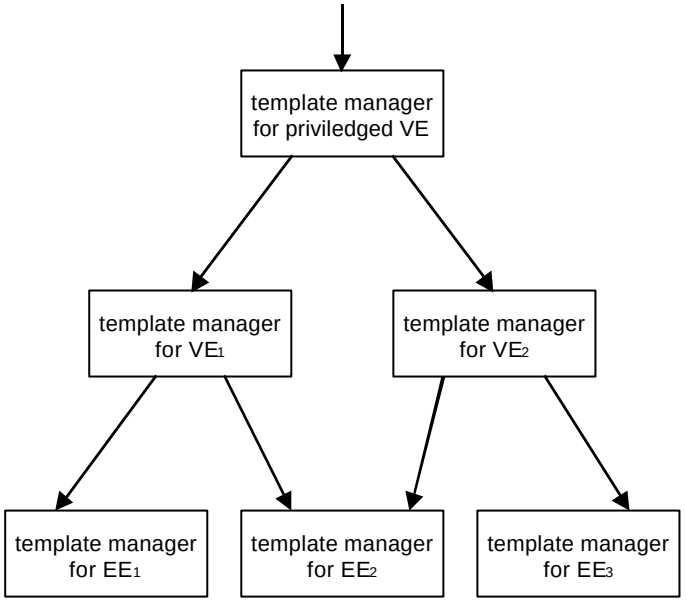


Figure 4-3: Hierarchy of template managers with respect to VEs and EEs.

The process of the installation begins with the examination of the description. After the class of the component manager is known it will be instantiated and initialised. The component manager can now use an EE specific API to reflect the installation inside the EE. The name of the template is returned as a result of the installation process. This name can be used to get the reference to the respective component manager. Additionally the component manager is registered at the template manager so it can be retrieved when instances of components need to be created later.

When a template is de-installed 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.

4.5 Specific Component Managers

While template managers are only specific to the execution environment in which templates should be eventually installed the component managers are even more specific. For each type of components there needs to be a component manager. When a particular type of resource is represented by a type of component (e.g. a process represented by a component) there has to be the appropriate resource manager (e.g. a process manager). In order to define a manager one should specify the additional interfaces and operations if any, the properties supported for resource creation, and the dimensions and units supported for monitoring resources if applicable.

There are two special managers: one for managing virtual environments and one for managing execution environments, other important managers are channel manager for demultiplexing packets to active services, traffic manager for managing traffic of a virtual environment, and the security manager.

4.5.1 Virtual Environment Manager

A manager for virtual environments is used to create and destroy virtual environments (VEs). When a VE is created the associated resources have to be specified. The VE manager will create all the

requested resources using the appropriate resource managers. When the virtual environment is activated the VE manager will activate all the previously created resources. The VE manager has to ensure that the resources consumed by components created inside the VE will not exceed the overall VE quota.

The minimal resources, which have to be specified for a virtual environment, are a suitable execution environment needed for running services and a channel needed for demultiplexing packets to service instances.

4.5.2 Execution Environment Manager

A manager for execution environments is used to create and destroy execution environments (EEs). When an EE is created the type and the associated resources have to be specified. The EE manager can use other managers (like process managers) to create an environment of the requested type. There may be cases where only one instance of an EE can exist like for kernel environments.

An EE can be assigned to one VE exclusively or it can be shared among VEs. In the latter case the components executing in the EE have to be charged for resource consumption individually to the appropriate VE.

5 DEMULTIPLEXING

5.1 Introduction

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 if an 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.

5.2 Survey and Requirement Analysis

At first, we survey related works and then define requirements for FAIN active packets and demultiplexing.

5.2.1 Survey of related work

5.2.1.1 ANEP (Active Packet Encapsulation Protocol)

Figure 5-1 shows ANEP packet format. ANEP is currently used in the ABONE (Active network backbone)

- **Version**

  It specifies a version of header format in use. This field is 8 bits long.

- **Flags**

  It specifies how to handle packets when a node does not recognize the type ID. This field is 8 bits long. But most significant bit (MSB) is only 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. In the current version, the MSB of flag's field is only used.

- **Type ID**

  It specifies evaluation environment of the message. This field is 16 bits long. The IDs for public use are under the authority of the Active Networks Assigned Number Authority (ANANA). Currently, 140 numbers are assigned by the ANANA.

- **ANEP Header Length**

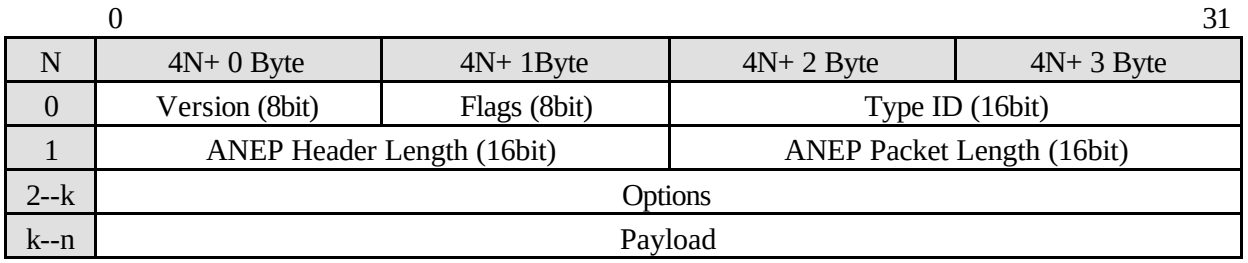
  It specifies the length of ANEP header in 32 bit words. This field is 16 bits long.

- **ANEP Packet Length**

  It specifies the length of ANEP packet in 32 bit words. This field is 16 bits long.

- **Options**

  It includes type, length and value of option.



	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--k	Options			
k--n	Payload			

Figure 5-1: ANEP Packet Format

Figure 5-2 shows ANEP option format.

- **FLG**

  It specifies how to deal with the option data. This field is 2 bits long. If the value of bit 0 is one, the options are only meaningful inside the specified evaluation environment. In addition, if the value of bit 1 is zero, option data is ignored and processing is continued. If the value of bit 1 is one, the packet is discarded.

- **Option Type**

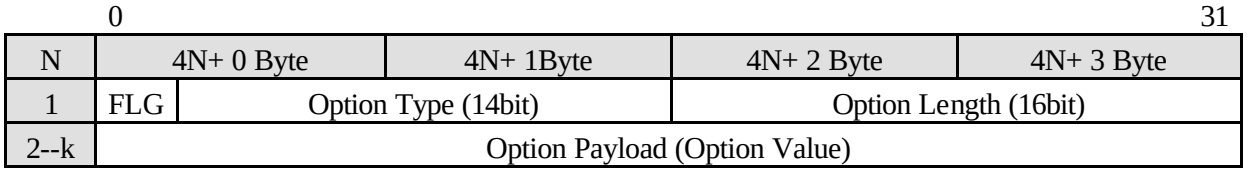
  It specifies a type of option. This field is 14 bits long. Currently four types are defined. 1)Source Identifier, 2)Destination Identifier, 3)Integrity Checksum, 4)N/N Authentication are defined.

- **Option Length**

  It specifies length of option data. This field is 16 bits long.

- **Option Payload**

  It includes the data of option.



	0			31
N	4N+ 0 Byte	4N+ 1Byte	4N+ 2 Byte	4N+ 3 Byte
1	FLG	Option Type (14bit)	Option Length (16bit)	
2--k	Option Payload (Option Value)			

Figure 5-2: ANEP Option Format

5.2.1.2 SAPF (Simple Active Packet Format)

Figure 5-3 shows a SAPF packet format. SAPF is simply composed of version and sixty-three(63) bits selector.

- **V (Version)**

  It specifies version of header format in use. This field is 1 bit long.

- **Selector**

  It specifies a value for selecting a packet handler. This field is 63 bits long.

	0			31
N	4N+ 0 Byte	4N+ 1Byte	4N+ 2 Byte	4N+ 3 Byte
0	V	Selector		
1	(63bit)			
2	Payload			

Figure 5-3: SAPF Packet Format

5.2.1.3 Berkeley Packet Filter in BSD OS

Figure 5-4 depicts a diagram of Berkeley Packet Filter(BPF). In this filter, when packet data are received at the network link driver, they are usually sent to a normal protocol stack. But when BPF is listening on this interface, the network link driver transmitted to the BPF filter at first. After that, a user-defined filter decides whether packet data are to be accepted or not. Then the packet data are transmitted to a proper application.

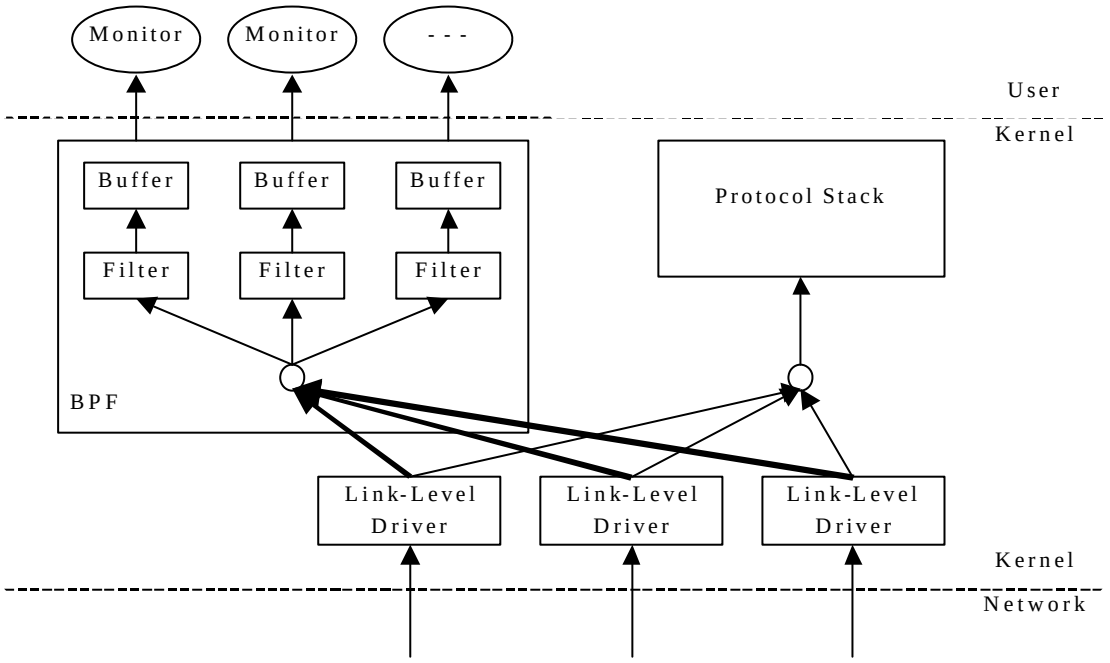


Figure 5-4: Berkeley Packet Filter

5.2.1.4 NetFilter in Linux OS

Figure 5-5 depicts a diagram of NetFilter. The Netfilter is currently used in the Linux operating system. It has five hooks. An incoming packet enters Hook-1(NF_IP_PRE_Routing) at first. For example, the Hook-1 is used for network address translation(NAT). After that, the packet is classified at a routing module-1, and routed either to a local process or forwarded to another host. When the packet is routed to a local process, it enters Hook-2(NF_IP_LOCAL_IN), before being transmitted to the local process. When the packet is forwarded to another host, it enters Hook-3(NF_IP_FOEWARD) and then enters Hook-4(NF_IP_POST_ROUTING). When a packet is created in a local process, it enters Hook-5(NF_IP_LOCAL_OUT) and then enters to a routing module-2. The packet is routed to the proper output port at the routing module-2 and is transmitted to outside network. In the NetFilter, Iptables is used for filtering a packet. The Iptables uses Hook-2, Hook-3 and Hook-5. Therefore a user program can control a packet flow by the Iptables.

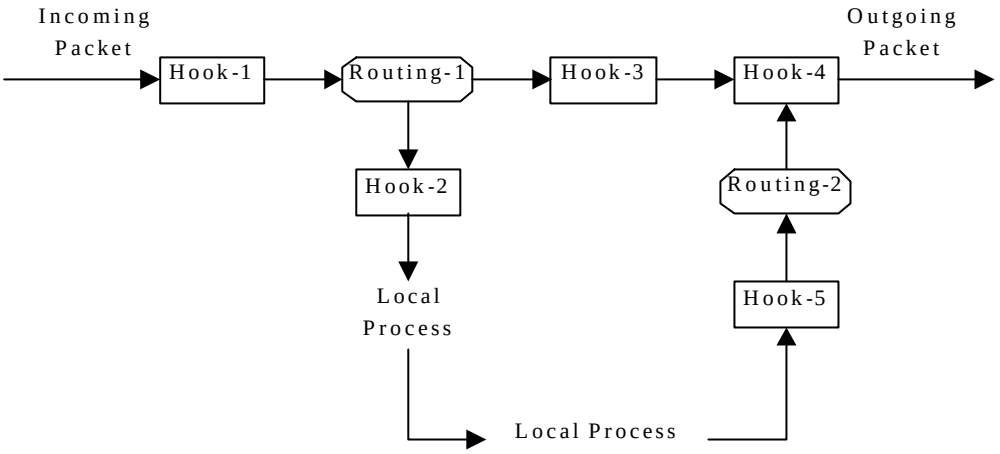


Figure 5-5: Netfilter

5.2.2 Requirement for Demultiplexing

5.2.2.1 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.

5.2.2.2 Requirement for Demultiplexing Mechanism

- Demultiplexing mechanism must deal with identifier of VE, EE type of active packet for distinguish of 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.

5.3 Demultiplexing Framework

The packet data are delivered to a proper VE or service by a demultiplexing function. The packet data include both ANEP(Active Network Encapsulation Protocol) packet and other data packet. The ANEP packet delivers active packet data and the other packet delivers not active data but data for being processed. Figure 5-6 depicts a block diagram of packet data delivery.

Active packet data (ANEP) delivery: (1) At first a VE/EE requests a channel manager to create a new active channel for receiving ANEP packet data. (2) The channel manager creates an active channel. (3, 4) Then the VE/EE sets a filter condition to the network(Netfilter) through the active cannel. The filter condition contains which ANEP packet data should be sent to the VE/EE. (5) The Netfilter transmits ANEP packet data to the channel manager. (6) The channel manager checks a security of the ANEP packet before sending it to a proper VE/EE. (7, 8) After executing the security check, the channel manager sends ANEP packet data to the proper VE/EE through the proper active channel. (9) If there is ANEP packet data for sending to another node, the VE/EE sends ANEP packet data to the proper active channel. (10) The active channel executes the security check before sending

the ANEP packet data to the outside network. (11, 12) After security checking, the active channel transmits ANEP packet data to the outside network by socket through the channel manager.

Non active packet data delivery: (13) At first a service requests the channel manager to create a new data channel for receiving data packet. (14) The channel manager creates a data channel. (15, 16) Then the service sets the filter condition to the Netfilter through the data cannel. The filter condition contains which data packet should be sent to the service. (17) The Netfilter transmits data packet to the channel manager. (18, 19) The channel manager sends data packet to the proper service through the proper data channel. (20) If there is data packet for sending to another node, the service sends data packet to the proper data channel. (21, 22) The data channel transmits data packet to the outside network by socket through the channel manager.

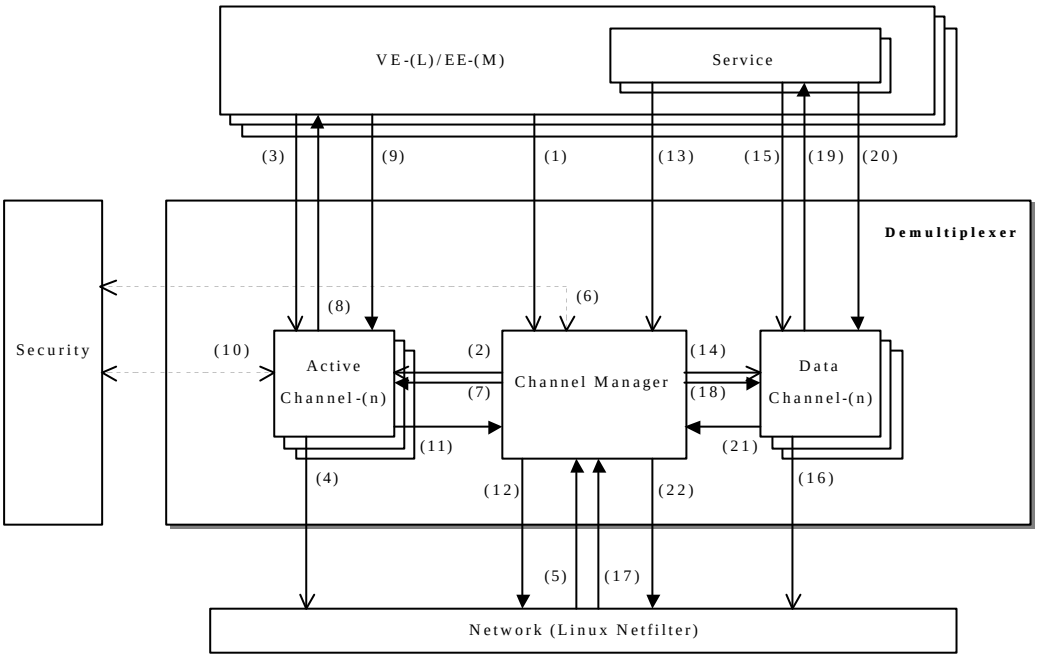


Figure 5-6: Block Diagram of Packet Delivery

5.3.1 Active Channel

The active channel is used for transmitting ANEP packet data from the Netfilter to a VE/EE. The active channel is created for each combination of a VE-ID and an EE-ID as shown in the Table 5-1. Therefore the VE/EE has to inform the combination to the channel manager for creating a new active channel.

Table 5-1: Relation between execution environment and Active Channel

No.	VE-ID	EE-ID	Active Channel
1	1	1	active_Ch-11
2	1	2	active_Ch-12
3	2	1	active_Ch-21
- - -	- - -	- - -	- - -
n	L	M	active_Ch-(LM)

5.3.2 Data Channel

The data channel is used for transmitting non- ANEP packet data from the Netfilter to a service. The data channel is created for each combination of a source IP address, a destination IP address, a protocol number, a source port number and a destination port number as shown in the Table 5-2. Therefore a service has to inform the data flow identifier to the channel manager for creating a new data channel.

Table 5-2: Relation between data flow and data channel

No.	Source_IP	Dest_IP	Protocol	Source_Port	Dest_POrt	Data Ch
1	sip-1	dip-1	p-1	sp-1	dp-1	dCh-1
2	sip-1	dip-1	p-1	sp-1	dp-2	dCh-2
- - -	- - -	- - -	- - -	- - -	- - -	- - -
m	i	j	k	l	m	dCh-(ijklm)

6 SECURITY

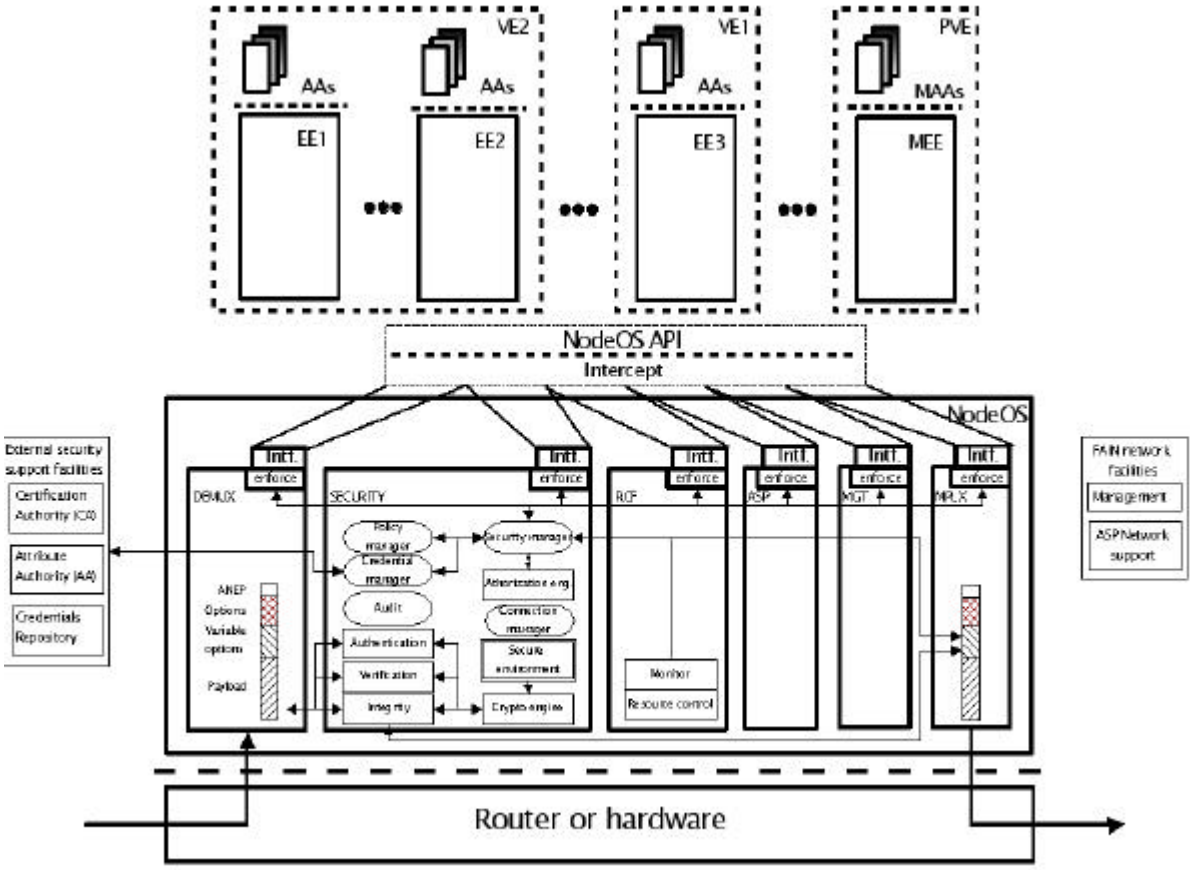


Figure 6-1: Revised security architecture.

The revised security architecture is depicted in Figure 6-1. The original security architecture in D2 has been enhanced with two new entities; the Security manager and the Connection manager, which are described below.

6.1 Introduction of the Security manager

One of the visible changes to the architecture is adding of the security manager to the architecture. Security manager was added to the architecture because of two reasons. First is that we feel that the approach with only one authorization engine wasn't flexible enough. There can be one or more different authorization engines in the system, which provide authorization decision. There is also a need in some cases with aim to get single authorization decision to combine one or more authorization decisions in single one. Case is obvious if we have to decide on the bases of two different security policies, which have different root of trust in the sense of the Keynote, trust management system. The second reason is that we needed an abstraction in the system that can enable us to export security policy and credentials interfaces to the rest of the system. The following picture shows the operation of the security architecture while providing authorization decision.

Security manager enables larger degree of flexibility then previous design. It enables us to keep the partial authorization decisions and combine them in latter stage to return complete authorization decision. In the system is an entity that is responsible for finding call related credentials and security policies. In the later stages of the project it will enable us to provide additional services like caching of authorization decisions and revoking of old authorization decisions.

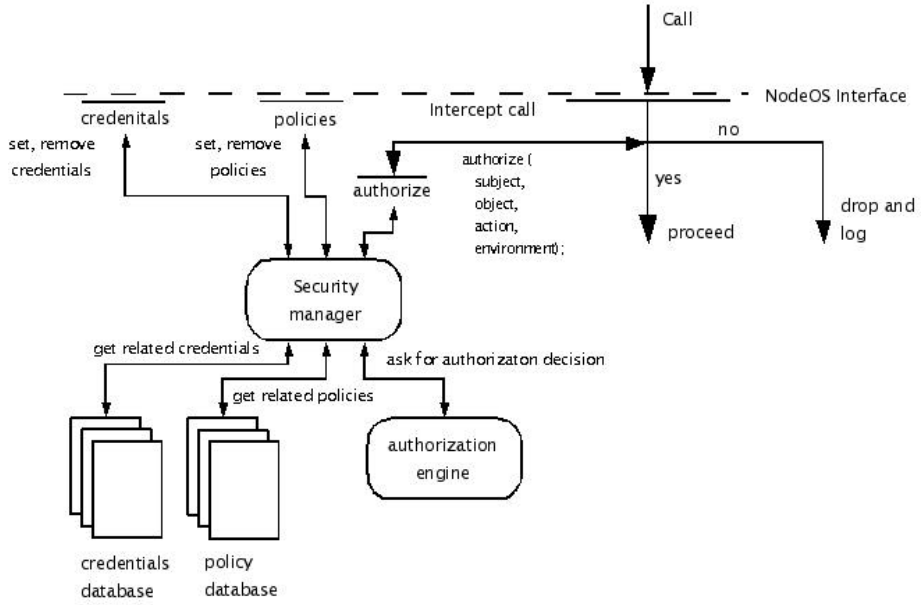


Figure 6-2: Security Manager interfaces.

Related to security manager our efforts mainly focused on generalising the internal authorization interface. This interface is available to the other NodeOS components for providing authorization decisions. While the interception of the call is interface specific, the authorization decision, seen as a service, should have same interface as for all interceptions. We have generalized the interface in the call with following parameters: subject, object, action and environment. While subject and object are obvious in this context, action means the methods defined in the interface, like file related read, write or append operation. Environment is sequence of statements, which describe the environment of the call like time or requested bandwidth.

6.2 Role of the Connection Manager

Though Connection Manager is new component in the security architecture its functionality is only extending Security Environment Manager as stated in D2. Connection Manager role is to take care for connections with neighbour Active Nodes and to provide interfaces, which can be used to manage existent and to define new connections. Connections are used to provide Hop-By-Hop data integrity service and are defined with following parameters: Key Identifier, that uniquely identifies the connection, symmetric key, which is stored in secure environment, identification of a symmetric algorithm used for this connection, time validity of the connection defined with start and end time, highest sequence number of the packet exchanged over the connection, and identification of the in or out channel protocol and addresses. These parameters are used to control the lifetime of the connection and to build or verify Hop-by-Hop integrity option as defined in appendix (A.1.4.1), where connection parameters are further explained.

Connections can be managed dynamically or via management system. At this stage security architecture supports management through management system, but we are working on dynamic connection management.

6.3 Operation of the security architecture regarding the security options

ANEP related security options have to be parsed and understood on every node the active packet passes. Parsed ANEP packet with options as defined in previous sections becomes on the node an object of the form as shown in Figure 6-4. The picture shows the four-step process to get the active packet related security context. Parsing of the packet gives basic parts of the packet object. V represents ANEP version used, F presents flags that control the node behaviour when the TypeId,

identification of the particular execution environment, is not known on the node. Options are sequence of options that are in the header but are not directly related to security architecture (besides resource vector). Private and ignored options are result of the parsing process and flags as defined in the ANEP options. Variable option as a security consequence is separated because it represents variable parts of users data or code. Static parts of user data/code are in the payload.

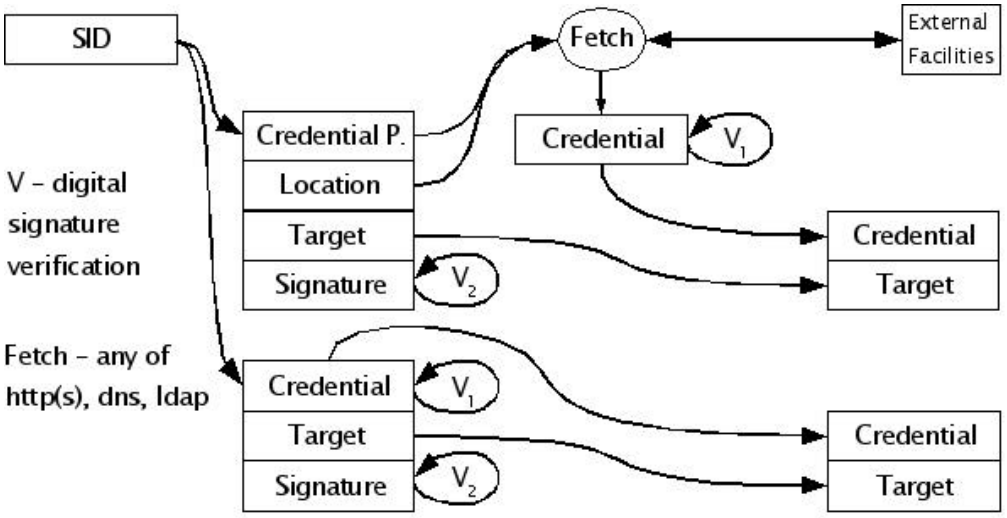


Figure 6-3: Credentials processing.

Credential options as defined in appendix (10.1.4.2), have to be parsed again to get security related fields. There can be one or more credentials options in the packet, each bind to the packet via digital signature mechanisms. The process of verifying and fetching credentials is shown in picture 5-2. Credentials can be already packed in the credential option or can be referenced in the option. The upper case explains the needed steps to fetch credential and to verify the credential itself. Credentials can be X.509 certificates, X.509 attribute certificates or KeyNote credentials. Authority that the node trusts digitally signs all such credentials. For first phase implementation we assume direct trust between such authority and nodes. With so gathered verified certificate or credential we can verify the digital signature in the credential option. Digital signature covers the credential option itself, the payload and static options of the packet (VE and EE Id). If the verification can be done, we get the credential, which can be used to provide security context on the node. Similar is the procedure in the case of in line credential but without intermediate step of fetching the credential. Both options are provided because the credentials can be big in size. The fetched credentials can be cached on the node.

Digital signature in the Credential Option is result of cryptographic operation on the parts of the packet that the signature covers with usage of the senders' private key. In the case of X.509 certificates the private key used must form a key pair with the public key in the certificate. In the case of attribute X.509 certificate the public key with which the digital signature can be verified is obtained from the X.509 certificate which is referenced in the attribute certificate (HOLDER field). In the case of Keynote credential the Licensee field must be a public key of the sender.

SID in the picture is unique pointer to the active packet in the node. It helps us track the packet through the node from demultiplexing the packet-to-packet processing and sending it toward next hop.

The last step related to credentials is resolving from the credentials either users identity, group role or public key. Public key is related to the trust management systems like KeyNote, where principal authorization information can be related to the principal public key. The resolved identifier has to be meaningful for already installed security policies on the node to be able to provide authorization decision based on this identifier.

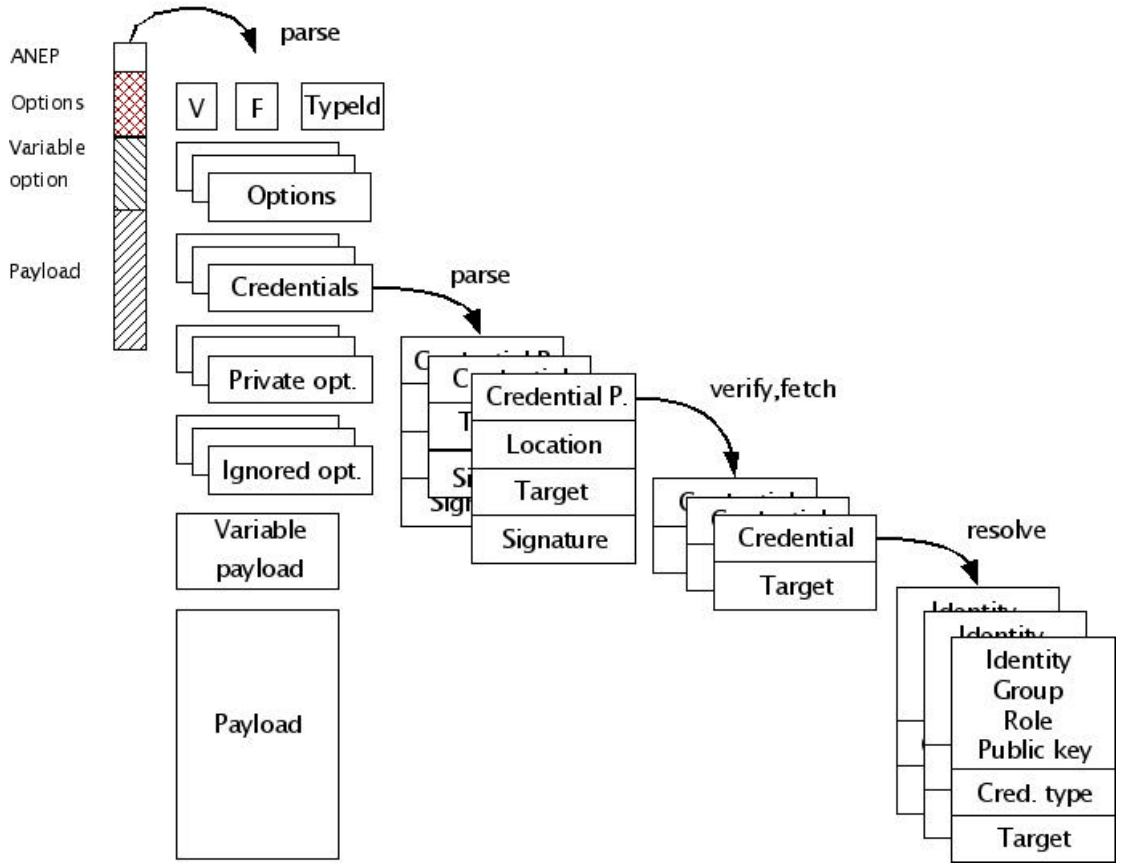


Figure 6-4: Parsed ANEP packet and Credential Option resolving.

6.4 Security related interfaces to other node submodules

Security architecture doesn't work alone in the active node. It has to have meaningful interaction with other node submodules. For this purpose we have defined a set of interfaces for each node submodule.

6.4.1 Interfaces between SA and DEMUX

Interface between DEMUX and SA is first critical point of the architecture. At this point the hop-by-hop integrity of the active packet has to be checked and packet, which can be seen as request, authenticated and if needed authorized to access destination channel. Because this process should not be driven by DEMUX, we have identified only one interface, which passes needed data to security area, which does all needed checks.

When the packet leaves the node there is a problem of building the packet and provisioning of the integrity service for the packet for the next hop. This can be build only with data in SA, so the security area will build the entire packet on the ANEP level and up, and return it to the sending component. After that sending component can add appropriate lower level protocol information to this data.

6.4.2 Interface between SA and ASP

Interface between SA and ASP is related to the verification of active code. As defined in D2, the first approach that we will use in FAIN is provided by digital signature mechanisms. Therefore we have proposed the code certificate, which is built in the same sense as KeyNote credentials. It consists of certificate version, authorizer, licensee (in this case code hash), conditions and digital signature. Code certificate is signed by private key of the authorizer. Of course the authorizer has to be trusted on the node. This means that the public key of the authorizer has to be meaningfully defined as such in the node security policy and available on the node.

6.4.3 General SA interfaces

Security architecture offers set of general interfaces that can be used by other node sub-modules and Virtual Environments. We count among these interfaces already mention Security Manager interfaces, namely credential and policy interface, Connection Manager interface for managing connections with the neighbour Active Nodes and internal authorization interface. Interface for providing authorization decision and interfaces to set or remove policies as can be seen on Figure 6-2 are general interfaces of the security architecture. Credential interface can be used in cases when credentials are not set via active packets but through management system (VE Manager). Primary source of security policies is management system that can use security architecture policy interface for this purpose. Both interfaces in addition to Connection Manager interface are subject to authorization check.

7 RESOURCE CONTROL FRAMEWORK (RCF)

7.1 Introduction

During the first year of the project, the resource control framework activity identified a certain number of key resources, mostly resources manipulated at the operating system level, that have been classified into physical and logical resources. It also identified a set of requirements on the consumption of resources, defined an element-based architecture for a resource control module, and designed the elements of this resource control module. This initial resource control framework provided system foundations for resource control services, and a flat architectural model of resources.

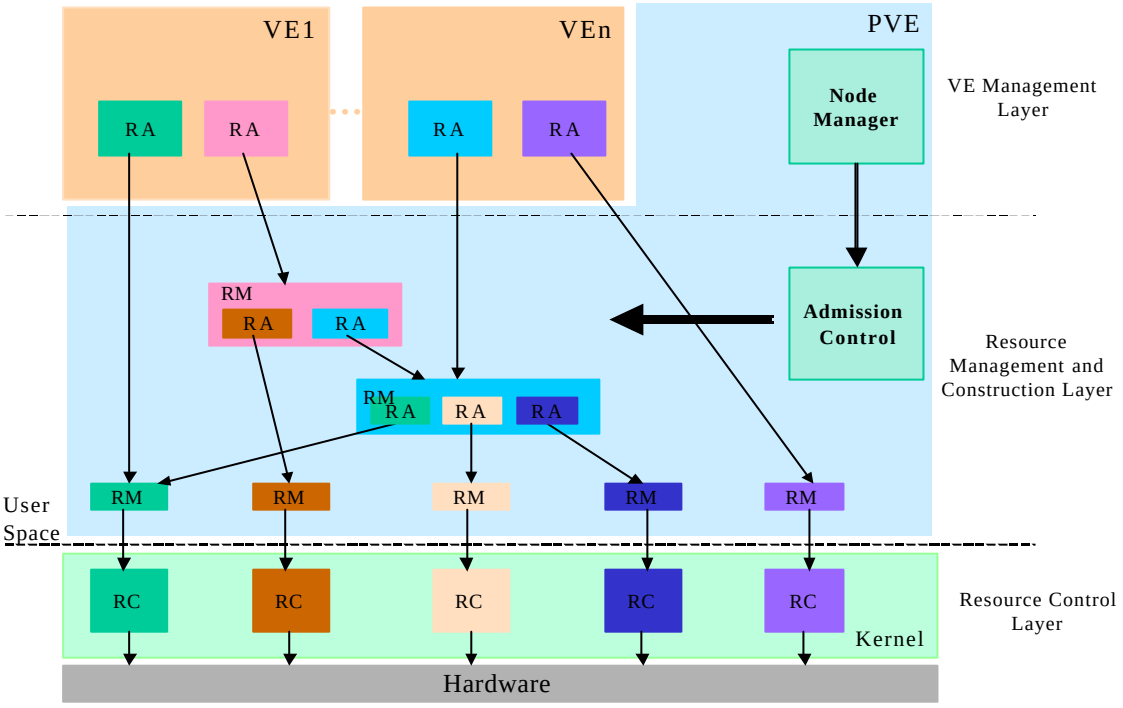


Figure 7-1: RCF reference architecture

The resource control activities during the second year of the project extended the initial proposal with the definition of a higher level of resources (medium and coarse grained resources) needed by the resource control and management functions of the VEs, and establishes the foundations of a hierarchical resource control system, that generalises resource the definition of resources at various levels of services: hardware level, OS, EEs, VEs, and application levels. Such a resource control and management system can be used to allow resource manipulations at different levels of services, and can act as a client of services provided by the resource control system defined during year 1. The proposals of the previous years can be seen as examples of resource control service personalities, that can rely on functions of the initial resource control system. Other resource control systems can be defined and implemented as well. The proposed resource control systems are intended to be used as supporting services for the implementation of several node components, including node management and VE components, network service and application components.

7.2 Resource control architecture and mechanisms

7.2.1 Revised RCF architecture

We define a resource control architecture for the management of virtual environments. It consists of resource controllers (RC), resources managers (RM) and resource abstractions (RA) organized as it is

shown in Figure 7-1. Service components RCs, RMs and RAs are higher-level service components (compared to those defined in D2) and can be implemented using the resource elements that have been identified in deliverable D2.

The resource control architecture presented in Figure 7-1 includes resources positioned at the hardware level, the resource control layer in the kernel space, resource positioned at the user space and at the VE space. Resources at the user and VE spaces in particular are high level resources, that are implemented using low level system and hardware resources and mechanisms and algorithms that combine these resources. The elements of the resource control architecture are defined as follows:

Resource Controller (RC): is a component, which is running in the Kernel Space of the node, and it is responsible for the actual control of the real resources of the Active Node. An 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. Netfilter Framework, Traffic Control Framework). Every RC has a control interface that allows its runtime configuration, which include the allocation and monitoring of the resources.

Resource Manager (RM): For every RC in the Kernel Space an RM exists in the User Space, which is responsible for the configuration of the corresponding RC and the creation of Resource Abstractions of the resources that are responsible for, for every consumer of a particular resource. These RMs are the elementary RMs in the sense that each one of them is responsible for only one physical resource. RMs also exists for the construction and the control of complex resources, which are the combination of more than one lower level resources. The RMs are consumers of the lower level resources, which could be other RMs, in the sense that they bond specific capacity from them, and at the same time are resources for the higher level RMs and the RAs. Among others the RMs are responsible for the Admission Control of the incoming requests for new allocations and for the realization of the allocation either by the configuration of the corresponding RC or by the transmission of the request to the appropriate lower level RMs.

Resource Abstraction (RA): The RAs are specific service components, which are bound up with specific amount of resources. The RAs are the end consumers from the RCF point of view. For every RA, a corresponding RM should exist in the Resource Management Layer but a RM could be responsible for many RAs. The RAs provide an I/Fs to the VE or the complicate RMs that belong, in order to be able to use their capacity.

7.2.2 RCF Design

The I/Fs are based on the FHG's Node Management framework as the RMs and the RAs can be considered as part of it. The IDL I/Fs that RCF exports to the other frameworks can be found at the appendix of this document.

7.2.2.1 Design approach and overview

We consider that system and hardware resources, services and components of an active node are organized into a hierarchy of resources, which will be used for control and management purpose. This hierarchical organization allows control and management operations at various levels of details, ranging from a simple node component (e.g. a buffer in a forwarding map) to complex node components (e.g. system service software on a control processor), using object oriented design and system engineering tool (e.g. UML, CORBA). Control and management in such a system can be done using APIs provided at different levels of resources, depending on applications and the capabilities of each node. Figure 7-2 shows an example description of the components of an active node. It only illustrates the structure of node resources according to a resource hierarchy, that ranges from – low level – elementary resources to complex resources usable by node and network management services, or application services running outside the network. The resource hierarchy in Figure 7-2 is just an example, which can be modified and adapted to specific needs. This example illustrates that:

- Resources in a switch or router consist of resources used by its switch fabric(s), its multiplexer(s), its forwarding map(s) and its control processor(s).

- Resources in a multiplexer consist of resources used by its buffer manager(s), its buffer(s) and its schedulers.
- Software resources in a control processor consist of resources used by its kernel, its control threads, its event handlers, and VE engines.
- Resources involved in a VE engine consist of resources used by its EE(s), its RAs, and its distributed object servers (e.g. CORBA servers).

This resource control and management model generalized the flat model resource model presented in D2, that deals mainly with system and hardware resources, so as to allow control and management operations at different levels of details, ranging from the simplest (controllable and manageable) node resources to sophisticated resources, e.g. those used by system and application software.

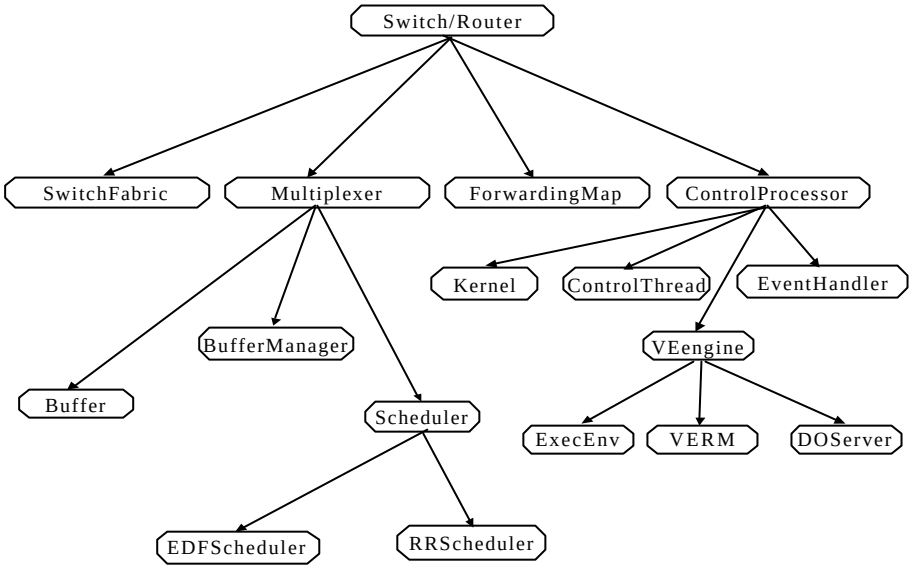


Figure 7-2: Example resource hierarchy in an active node

7.2.2.2 Resource control operations in VE management

A VE is a potential consumer of almost all types of resources in a node. It can use control processing and memory resources, packet forwarding resources (flows, buffers, schedulers, …) and possibly management resources. From a software engineering point of view, resources are special components used in VEs that need to be controlled and managed to meet the requirements of applications running in a VE. During the creation of a VE, a given amount of resources of each resource class (processing and memory resources, packet forwarding and flow resources, management resources) is allocated on request of the organization that owns this VE. These resources will be multiplexed between services supported by that VE, with different kind of resource consumption requirements. For example:

- An application service can require that a fixed amount of resources of each class that it will consume should be allocated for a relatively short lifetime (compared to that of other application or the VE). When such an application terminates, its resources will become available to be allocated to another (incoming or already running) application. Such a resource allocation can be done using a static admission control mechanism that checks the availability of resources before service creation.
- Another application service can be such that it is impossible to determine the amount of resources needed for its execution, before its creation. For example, this is the case of many end-to-end network service applications that need QoS guarantees. In such a case dynamic resource control and monitoring mechanisms are needed to ensure the availability of resources during its lifetime. Resources for such applications can be provided using run-time resource control mechanisms (this

is the case of applications with e.g. strong guarantees related to real-time or reliability) and/or resource monitoring techniques (this is the case of applications with e.g. real-time or reliability upper and lower bonds). Such applications require an admission control check for their creation, followed by a resource control and/or a monitoring mechanism during all their lifetime.

An RA, which is a VE component that deals with resource management and control needs, is used to deal with resource availability requirements within a VE. One can build several specialized RA (e.g. for different classes or categories of resources, for different applications, etc), or build a unique RA that deals with resource guarantees for a whole VE. An RA will implement parts or every one of resource management interfaces identified in the node management framework, which are:

- A resource control interface.
- A resource monitor interface.

The implementation of some of these interfaces can use run-time resource control operations: for example, a flow manager can use such operations to request for the creation of resource control and monitoring services, that will guarantee the adequate level of availability of resources during the lifetime of a flow with QoS constraints. Immediately after the successful creation of a flow, resource control and monitoring services will be operating in parallel with the delivery of packets of that flow, to guarantee the availability of resources for an adequate handling of packets. The flow manager will also request for the destruction of control and monitoring services for that flow, after it terminates.

7.2.2.3 Run-time resource control service for the management of VE resources

This section presents the principles of a run-time resource control system that can be used for the management of VE resources and the control of traffic with QoS requirements. Such a system can be used to implement some of the interfaces of an RA component, e.g. flow management interfaces, resource monitor interfaces (possibly in collaboration with a resource monitoring service). We consider a special RA component, that manages and controls the usage of packet forwarding resources used by a VE: such a RA component is responsible for the initial allocation and creation of packet forwarding resources needed by a VE, the freeing of these resources at the end of their usage, and their multiplexing among parallel and concurrent services.

The RA creates a run-time control service during its initialisation. Upon its creation, a unique interface to the run-time control system is returned and published for control operations by local client applications and also by remote clients. All the interfaces of the run-time control system will be supported by a DOC platform (e.g. a CORBA system) to enable invocations from different address spaces, including from a remote node. At the end of the lifetime of the RA, the run-time control service will be released, just before the destruction of the RA itself.

The run-time control system is used to control resources used by individual traffic as follows:

- Just before the VE participates in traffic, the client application asks for the creation of an entry, using the run-time control interface. The entry provides means to accept the incoming traffic request, which will be confirmed (or not), depending of the current resource usage.
- When a traffic request is accepted, an entry is created. Upon the creation of the entry, an interface to that entry is returned to the client application for subsequent packet transmissions.
- An entry interface provides operations to access information required for the transmission of data packets (port numbers, buffer addresses, etc.). These information will be used to request for the transmission of packets belonging to the traffic using that entry. Before sending a sequence of packets, the client application asks for the creation of a controller for that sequence, which is a logical resource that will guarantee the availability of packet forwarding resources for the incoming sequence. A controller maintains resources needed for the transmission of packet from one end to another, within the same node. A controller also provides interfaces for its control, usage and management. When the sequence in finished, the controller can be destroyed, stored for

further usage, or allocated to the transmission of another sequence, depending on specific implementations and traffic conditions.

- As a summary, the run-time control system supports three types of objects: a unique access object, one or many entries, and one or many controllers. The access interface serves as a single reception point for all incoming traffic request. Entries serve as reception points for individual traffics (e.g. a communication session). Finally, controllers are use to enforce QoS guarantees for sequences of packets with a given QoS requirement.

It is the role of the RA to guarantee to proper functioning of the run-time control system as a whole. To do this, the RA could perform some management operations on the run-time control system, typically via a RM for that RA (in accordance of the reference architecture in Figure 7-1): add or remove packet forwarding resources, update the number of allowed entries and controllers to adapt the dimensions of the resource controllers, to actual traffic conditions. Such management operations can be realized within the code associated to one or many of the internal services and/or interfaces supported by the RA, possibly via one or many RM(s) for that RA.

This describes the principles of a run-time control service and its usage by a RA and client applications. The principles of this run-time system will be generalized and described more in details later in this RCF section, for a run-time control system that applies to virtually all active node and network services with QoS and flexible processing requirements.

7.2.3 Admission Control

7.2.3.1 Introduction

The FAIN active node aims to be an open environment where the user will deploy his network services by using a part of the node infrastructure. In order for the node to be able to support these services the user should have guaranteed access to the necessary resources. The resources that are needed from each user are allocated to a VE, which after its creation is available for the exclusive use of its owner. The creation of a new VE in the FAIN Node cannot always be accepted because of the finite amount of the resources. This requires the existence of an Admission Control mechanism within the RCF of the FAIN Node.

The Admission Control in the FAIN Node addresses a set of actions that should be taken by the RCF during the VE's creation phase (or during re-negotiation phase) in order to decide whether a VE creation request can 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 be violated. The final decision for the acceptance or not of the request for the creation of a new VE should take under consideration three factors, namely the unreserved resources, the needs of the new VE and finally the usage of the resources during the past period of time. In parallel the increase of the node's utilization should be achieve by the acceptance of as many VEs as possible.

7.2.3.2 Types of Allocations

The guarantees that the VEs have for access to the node's resources may vary due to the different needs that the services might have, mostly regarding the QoS requirements. So we can define different levels of allocation guarantees that the RCF should be able to support to the VEs. Namely we have three types of allocations:

- **Hard Allocation**: The VE that has hard allocation of a resource has guaranteed access to the whole allocated capacity, independently of the status of the node. In other words, the VE will have full access to this resource even if there is congestion in the node. Hard allocation is needed for hard real-time services with hard QoS constraints.

- **Soft Allocation**: When there are no strict QoS requirements and therefore resources requirements from the Services running in a VE, the VE could require only soft allocation for the resources. Soft allocation can be used by soft real-time services, which can provide an enhanced QoS without the need for hard guarantees.
- **No Allocation**: There are services that don't need any performance guarantees, and they perform only when there are unused resources in the node. These services are called "best effort", and even though they need resources to perform they don't need any guarantees and therefore no specific resource capacity to be allocated. When in a VE are executed only these kinds of services, there is a need to have access to the resources but there is no need to allocate some of them.

7.2.3.3 Admission Control Model

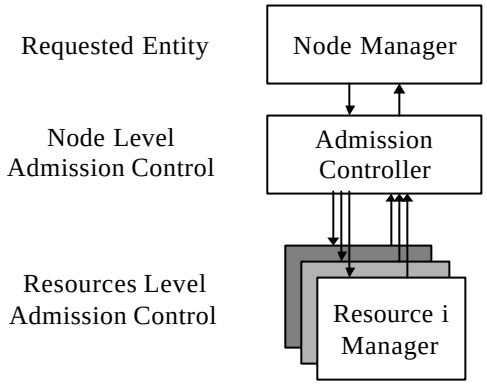


Figure 7-3: Admission control process

The admission control of whether a new VE will be admitted or not in the FAIN Node is the main responsibility of the RCF block together with the run time control and monitoring of the resources. In contrary to resource control and monitoring which are performed during the operational phase of the VEs, the admission control performs only during the creation or the renegotiation phase of the VEs.

The Admission Control functionality is the responsibility of a central object, the Admission Controller (AC), which is in close collaboration with the RMs. In fact the AC is responsible for the Admission Control in Node Level and the RMs are responsible for the Admission Control independently for their corresponding resources.

The creation of a VE is performed in two phases, namely the Pre-Allocation Phase and the Activation Phase.

On the Pre-Allocation Phase the Node Manager, which is responsible for the creation of the new VE, makes a request for Pre-Allocation to the AC. The request contains the profile of the resources that are needed for the new VE. When the AC receives the creation request, gets in contact with the appropriate RMs and requests the pre-allocation of the resources, and collects the RMs responses. If all the responses are positive, which means that there are enough capacity from every resource, responds positively to the Node Manager that the resources are pre-allocated and ready to be activated. On the contrary if one or more responses from the RMs are negative the response of the AC is negative too.

On the Activation Phase the Node Manager makes an Activation Request to the AC. Then AC makes requests for creation of the corresponding RAs for the new VE and gets as response the IDs of the RAs. After that the AC return the set of the IDs to the NM as the result of the activation.

Figure 7-3 shows the relationship of the AC with the RMs and the two levels in which the AC is taking place. The Node Level Admission Control examines the availability of the overall node and the Resource Level Admission Control the availability of the different Resources.

7.2.3.4 Admission Control Algorithms and Policies

In order for the RCF to decide whether or not the creation of a new VE can be accepted, is has to use admission control algorithms and policies basically to predict if the requirements of the new VE can be satisfied without violating the existing contracts.

The AC uses the Admission Control Policies in order to decide based on the replies of the RMs, whether or not it can admit the creation of a new VE. For example the during the Pre-Allocation phase that described in section 1.2.3.3, the AC is taking the decision based on a hard policy, that is that a request can be admit only if all the replies from the RMs are positive. Of course more tolerant and flexible policies can be used but for the time being, this seems to be the right choice for two reasons: it is simple to be implemented and it guarantees the satisfaction of the QoS constraints.

The Admission Control Algorithms are used by the RMs in order to decide whether an allocation request can be satisfied. There do exist many Admission Control algorithms in the bibliography. The choice of which algorithm will be used for each resource depends on the specific requirements that are derived from the nature of the resource, and mainly on which mechanism/algorithm is used for the share/control.

There are two basic approaches to admission control:

- *parameter-based*: computes the amount of the resource required to support a set of VEs given the VEs' characteristics
- *measurement-based*: relies on measurement of actual use of the resource in making admission decisions.

Parameter-based admission control algorithms can be analysed by formal methods. Measurement-based admission control algorithms can only be analysed through experiments on either real networks or a simulator.

7.3 Resource control services and applications

7.3.1 Principles of a distributed end-to-end network engineering support

7.3.1.1 Overview

We present the principles of DENES (Distributed End-to-end Network Engineering Support), and its relations with other FAIN nodes service components and frameworks. DENES is an active and programmable node service that provides a resource control support for the implementation of distributed control and management services. It can be used as a control basis to implement various kind of end-to-end network services, distributed management services and application-level services based on active and programmable networks. DENES addresses the issue of providing a network-wide QoS guarantees, by means of hierarchical resource control and computational behaviour adaptations.

7.3.1.2 Overall approach of resource control operations

7.3.1.2.1 Basic principles

The DENES system is built upon the following considerations:

- Resources available at a node are provided with software representations of their functions, which allow a local (centralized) control or management of these resources. The level of details at which resources are represented depend on the needs of applications, and can vary from one of the simplest to a very complex, detailed resource representation.
- The system that represents resources is called the *reflection system*, and is typically modelled and implemented with objects. For example, a reflection system can be built for a distributed object computing (DOC) platform (sometimes called a reflexive middleware) exposing APIs for control,

management and monitoring operations for the DOC platform, that could be use to modify their implementations. An example such facility is given the Java introspection mechanisms, that provides a foundation for discovering new objects, interface and system specifications, and thus, re-building parts of the platform itself. Services supporting the L interface of P1520 are in fact building a reflection system for the packet forwarding machinery, which allows controlling the usage of packet forwarding resources.

- Resources represented in a reflection system (*reflectors*) are accessible from other services and address spaces within the same node, and also from remote nodes. Accesses to some of the resources can be forbidden to some clients or limited, under some specific security or access control conditions. Accesses to resources are enabled by e.g., the use of a DOC platform, which provides interfaces for remote object invocations. An interface to a reflector can be used, for example, to capture the current state of the corresponding resource, to perform various operations related to its evolution, to monitor its usage, or even to modify its computational behaviour. The later kind of operations in particular allows developing and deploying new executable codes to be used for the computations associated to a reflector in replacement of existing computations, provided that some security and consistent resource usage conditions are met.
- An active node can support the execution of one or many DENES services, depending on implementation choices and the number and the planned usage of VEs (supported applications). A DENES enabled active node is intended to allow clients from the same node, homologue active nodes or from outside the network to perform individual node control operators, as a prerequisite for distributed, end-to-end network control operations that rely on the availability and the adequate behaviour of resources. For example, DENES engines can be used to select an admissible sub-network under some QoS conditions, so as to establish an adequate end-to-end transmission path.
- No specific assumption is made on the technology on which active nodes are based, a part from the fact that they are assumed to support open interfaces for programmability at more or less complex service levels (especially for packet forwarding functions), and general active network service components similar to those defined in and being used in FAIN, and that it is intended to perform flexible multimedia communication functions with QoS constraints. A node could be based on an ATM switch that implements Internet protocols, or Internet router technologies – including those intended to support next generations of internet service, e.g. base on IPv6, MPLS, and so on.

7.3.1.2.2 Service paradigm

A DENES engine operates for an entire node, one or many FAIN service spaces that make use of one or many VEs and EEs, depending on the partition of memory, computing and packet forwarding resources, and resource control implementation choices. We only require that resources controlled by a DENES engine belong to a unique referential for processing, memory management and transmission resources, in order to free a DENES engine from - non-trivial - concurrency, synchronization and resource state monitoring problems.

A DENES engine regularly reacts to service requests from clients, for the transmission of:

- a packet,
- a sequence of packets with a given QoS profile (an estimated constant QoS),
- or the packets of an entire service session.

Clients are local network and application service components, homologue active nodes or host systems operating outside the network. Requests are addressed through a control channel, in order to ask for resource availability before actual data communications. The DENES engines analyses resource availability for incoming requests, performs resources reservations if needed, and sets-up adequate activities and mechanisms in order to guarantee the incoming service contract when it is possible. In case the supporting node cannot satisfy the specified resource needs, the DENES engine notifies the client that it cannot satisfy the request, so that the client can try another node or reconsider its QoS.

7.3.1.2.3 Objectives, usage and interests

A client can request the transmission of a packet, a sequence of packets, or packets of an entire service session through a node. This client will ask a DENES engine for QoS guarantees before any effective data transfer. The results of such an inquiry can be one of the followings:

- YES, this node can guarantee the required QoS,
- NO, this node cannot guarantee the required QoS,
- UNDETERMINED, at this time it is impossible for this node to say whether or not it can guarantee the require QoS.

In the first case, the DENES engine guarantees that given the current resource conditions and the specified QoS requirements, the node can handle the incoming traffic and engages to respect the specified QoS contract. In the second case, the current resource conditions are such that the DENES engine is sure the incoming traffic cannot be handled with the required QoS. In the last case, the resource conditions and the QoS requirements are such that the node cannot determine if it can handle the incoming traffic or not. By resource conditions, we are talking about both the state of physical and logical resources to be used and their behaviour, which is encapsulated, typically with a hierarchical object-oriented reflection system, that presents local and remote invocation interfaces, and that can be modified at any time though active code injection mechanisms.

We aim at a DENES engine, which provides the same type of interfaces for all the requests in order to simplify APIs. Also, DENES engines must be able to handle active packets transmitted with requests, and interact with the corresponding node services components (e.g. the adequate EE) in order to activate the execution of new codes.

A request should contains all the information necessary for a DENES engine to decide on the possibility for nodes to handle the corresponding traffic, along with a specification of QoS constraints. Upon the reception of a request, a DENES engine identifies all the resources involved in such traffic, and analyses their availability (e.g. I/O ports to be used, for the forwarding of packets from one end of the node to another, schedulers, buffers and queues, etc.). In case it is impossible to satisfy the specified requirements (from a first analysis), a DENES engine might try to modify resources for a better service (e.g., try to allocate more memory for buffers, for a reliable traffic). Thus, we would like to be able to dynamically modify both the computational behaviour of resource and their size (especially concerning physical resources), at the limit as frequently as the occurrence of traffic requests. In the extreme case, a DENES engine can “install” an entire packet forwarding machinery transmitted through a request (under some security and consistency constraints), which will be used to ensure an incoming traffic.

A DENES engine provides a node service support for the deployment of flexible multimedia communication services with various QoS constraints. Using DENES, it is possible to implement distributed policy-based network management (e.g. PEPs and PDPs as DENES clients, where policies will be used to derivate QoS constraints), existing and next generations of Internet protocols, multicast services and applications with QoS constraints (reliability in particular), VPN, etc.

7.3.1.3 DENES service model and role in a node service architecture

A DENES engine provides the run-time support for objects running on the control system of an active network node. It supports the concurrent execution of objects that allow an access to node resource control functions, from both the outside world and other node service components. This is done via the use of public interfaces, that are made accessible to authorized dients during the lifecycle of theirs supporting objects. Objects running on a DENES engine allow clients to:

- Request for the initialisation of a new traffic with QoS and resource usage constraints.
- Control the use of resources, to guarantee their availability during traffic.

- Release or re-allocate currently used resources for another traffic. A DENES engine can act for several ongoing traffics concurrently.

Virtually, a DENES engine can provide resource control for the use of any kind of resource in a node. However, the current design focuses on control operations for the use of physical packet forwarding resources and their immediate software representation (reflectors, for programmability), for QoS guarantees and flexible packet forwarding systems. The current DENES design focuses on accessibility and resource control for packet forwarding resources, under QoS and resource modification constraints.

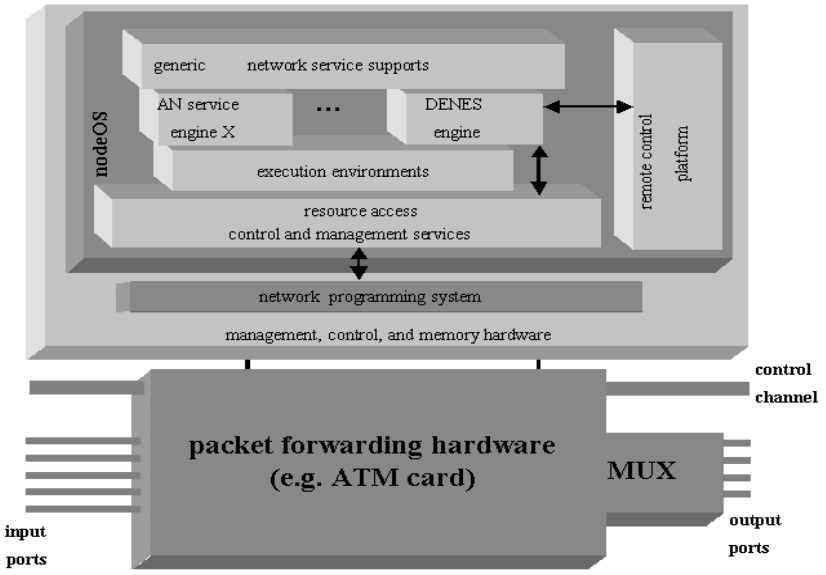


Figure 7-4: A DENES engine in an active node

Figure 7-4 illustrates the position of a general DENES engine in an active node, and its relations with other node services. It is intended to use services provided by active node execution environments, remote control platforms, local resource access control and management functions and possibly NodeOS primitives, control and management processor operations, packet forwarding hardware drivers, etc. Its target clients are the generic network services supports and applications, services components running outside the node, and possible other active node service components.

- Packet forwarding hardware drivers and primitive control and management control processor operations can be used to provide low-level resource related operations to the outside word in case these operations are not supported by the NodeOS.
- NodeOS primitives can be used to provide an external access to operating system primitives, especially those with active node constructs.
- Basically, the remote control platform is an object-oriented middleware, which allows remote object invocations on DENES supported objects. The term “control” here is related to the execution of a (sequential) application code on a remote system. In case a remote control platform is used by in the EE for example (e.g. an ORB-based EE), the same control platform can be shared.
- DENES engine is using EEs and other active node services for the execution request of new codes (e.g. replace services provided by a packet forwarding system), or to benefit from other active node related services (security, code lookup, code mobility, etc.).
- Generic network service components will use DENES engines to develop flexible end-to-end network service components, which control node resources to satisfy a required QoS.

7.3.2 DENES service architecture and component specification

7.3.2.1 Overview

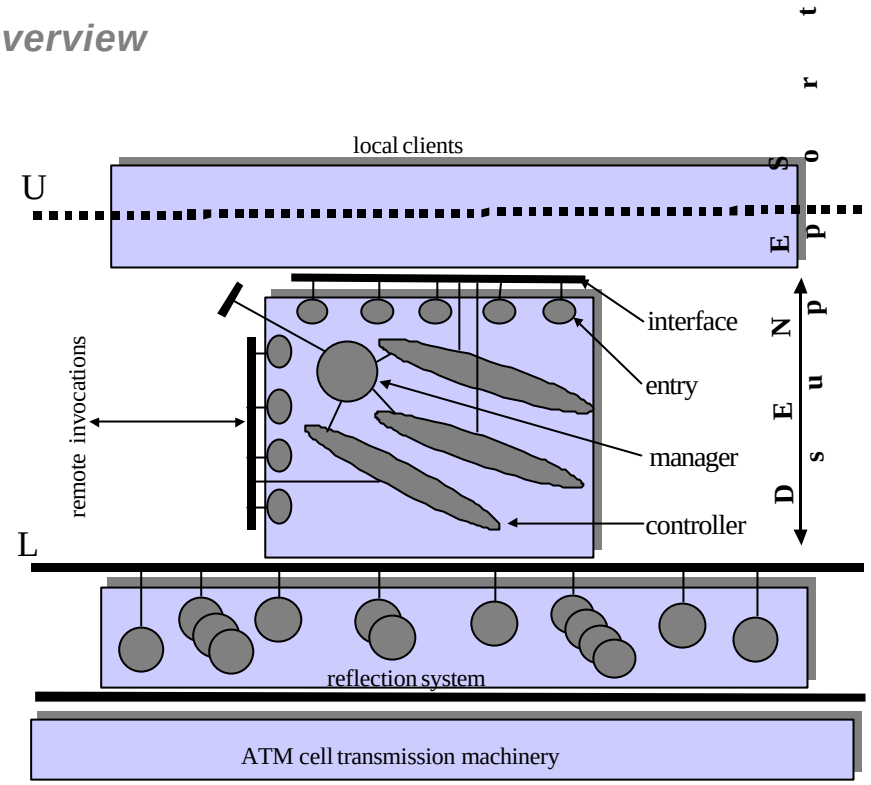


Figure 7-5: DENES service components and relations with P1520 service levels

Compared to the P1520 service model, a DENES service is roughly situated between the L interface and the U interface. Its level of service is equivalent to those of the U interface supporting services. We do not try to position here service provided by an EE or a remote control platform, which are in a transversal service plan and can be virtually used in any other P1520 service level. As it is presented in Figure 7-5, a DENES service typically uses services supporting an interface L (currently, a reflection system associated to a packet forwarding machinery, typically that of an ATM system), and offers U-level interfaces, for both local and remote accesses. Local access interfaces are used by local client components, while homologue nodes or application host systems use remote interfaces.

The differences between local and remote interfaces rely on the type of resources needed for their implementation. For example, remote interfaces are implemented using communications via a control channel and node I/O ports, while local interfaces are implemented using a shared memory, a Hoare monitor or any other local communication and synchronisation mechanism. Local DENES clients can be services supporting parts of the U interface and upper (V interface), i.e., typically generic routing services, end-to-end admission controllers and connection managers, services used by various network and application service models (RSVP, DiffServ, MPLS,), policy-based network management services. All these clients can be local (their local representatives) or remote clients (a remote representative, for the case of distributed services).

A DENES system supports three types of objects:

- ENTRIES: An engine can support several concurrent entries, which are objects providing an initial access point for individual traffics.
- traffic MANAGER: An engine supports a unique traffic manager, which provides an initial access point to all the clients of the DENES system.
- Traffic CONTROLLERS: An engine can support several concurrent traffic controllers, which provides a top level reflector of packet forwarding resources used by an individual traffic, and

which can be manipulated from the outside world.

In the following text, we describe some of the major roles identified for each of these objects. These roles can be extended and described in more details for an implementation.

7.3.2.2 Traffic manager

A traffic manager offers a unique access point for the use of resource control services. Before any interaction, a client asks the traffic manager for the creation of an entry, that will be used for further resource control related operations. Thus, a traffic manager interface provides an operation for creating entries. However, the destruction of an entry can be done through an operation at its interface (preferable solution), as well as through an operation at the traffic manager's interface. The destruction of an entry should interact with other DENES supporting objects (the traffic manager in particular) in order to suppress all the resource bindings previously established by its creation, and to release resources consumed by the entry. Another important task of a traffic manager is to collect and process control requests issued by clients of entries. A DENES engine supports the execution of a unique traffic manager object, which supports a public interface accessible by both local and remote clients. From an implementation point of view accesses to such an interface are possible with the use of a DOC platform that allows the publication of a unique reference to the interface. A prospective client can use a DENES engine if and only if it has a reference of the traffic manager's interface.

7.3.2.3 Entries

An entry offers an access point to packet forwarding resources, to be used by individual traffics. It is used to isolate endpoints used for an incoming packet flow. Before any data communication, a client asks for the creation (or the allocation) of an entry. At the end of the communication, the entry is destroyed or made free for use by another communication. The creation operation returns an interface to be used to control the entry. Whence created, an entry can be used by its creator, as well as by a third-party client, provided that this third party has a reference to its interface.

An entry abstraction provides a homogenous interface for both local and remote accesses to packet forwarding resources. It can be used independently of the origin of the request. It is also used to identify basic transmission resources (e.g. port numbers or memory addresses) for communications from an end-point to I/O lines of the packet forwarding machinery. Initially, a DENES engine is installed with no entry point. Entrees are created and destroyed dynamically, depending on the arrival of requests issued by clients.

7.3.2.4 Traffic controllers

Traffic controllers are objects encapsulating packet forwarding resources used to transmit packets, typically from one end of a node (input ports) to another (output ports). A traffic controller offers means to isolate and explicitly control resources used for the transmission of a packet or a sequence of them with a QoS profile. The level of details in control operations depend on the possibilities offered by the packet forwarding machinery. For example, a traffic controller can allow modifications on the computational behaviour of some of the resources it uses. A traffic controller is used to guarantee the QoS contract established before data transmission. For example, for reliable packet forwarding, the traffic controller should guarantee that:

- Queues used by I/O ports will always have enough places to avoid packet loss due to the lack of memory to handle incoming packets.
- Scheduling policies and algorithms used will not lead to problems with consequences similar to those of a packet loss.
- Bandwidth allocated at output lines will remain stable.

When an entry is created, the same client to send several transmission requests, until the entry is suppressed can use it. A client can also change the entry currently used for a service, for a better

adaptation to the QoS requirements expressed by the ongoing traffic. Transmission requests typically specify a certain number of packets (or cells, in case of an ATM traffic) to be transmitted, with fixed resource requirements. When such a request is received, a traffic controller is created (or adapted) so as to satisfy the requirements. A traffic controller offers an interface to modify its configuration (memory and processing resources used) and an interface to access certain state information and perform some control operations. The configuration and modification interface typically leads to changes on the size of internal buffers used by I/O ports and multiplexers, algorithms used in the management of queues, etc. Information necessary for the modification are sent with a control request on an entry point, as an active packet, before any configuration change.

7.3.3 Current design focus: reliable packet forwarding and related problems

7.3.3.1 General reliability problems

Packet loss in a packet switched network is mainly due to one of the following problems, as they are identified in:

- The architecture and the management of queues in routers: The number of queues per service and service policies and algorithms (FIFO, prioritised queuing, WQS,) has a great influence on the efficiency (and ultimately the reliability) of packet forwarding within a router.
- Variability in the size of data: ATM cells for example are of fixed size (53 bytes), while IP packets are of a variable size. While one can easily dimension resources for the transmission of data with a fixed size, it is generally hard to dimension resources for a reliable delivery, when the size of incoming packets varies.
- The amount of memory, the allocation policies and algorithms for memory in a router.
- Traffic variations dictated by application.
- Link or router failures and related fault-tolerance mechanisms.
- Bit rate errors.
- The number of hops (router to cross from one end to another).

7.3.3.2 Reliability within a node

Hardware and software in a node must deal with the following problems to allow reliable delivery within a node. Router failure problems are not taken into account, since they involve fault-tolerant computing problems, which are out of the scope of the current specification.

- Buffer management, including the calculation of the adequate size for buffers.
- The architecture and the functioning (mechanisms, algorithms) of waiting queues.
- Resource allocation policies, including processing, memory an I/O transmission resources.
- The size of memory used by the different components of the system, and the related local communication and synchronization mechanisms.

7.3.3.3 DENES engine for hard reliability

Such en engine allows clients to send packets that require a guaranteed delivery upon reception at a node. A node that accepts a packet (or a sequence of packets) engages to deliver this (or these) packet(s) at the corresponding output port(s) or local destination subsystem within a finite time. The responsibility of the DENES engine is not engaged in case of a node or I/O link failure. The current DENES specification and prototyping effort is focusing on this kind of reliable delivery

Other reliable delivery semantics can be specified and implemented as well, for example:

- DENES engines that engage to deliver packets within a certain loss probability interval.
- DENES engines that engage to deliver packets with a specified packet dropping protocol, which will be used to drop adequate packets in case of resource problems. The packet dropping protocol can be specific and transmitted by a given application, based on the content on data being transmitted.

This second type of reliable delivery can be taken into account by extensions of a DENES engine for hard reliable delivery.

7.3.4 Relations with FAIN node components and applications

7.3.4.1 Relations with the resource and admission control

7.3.4.1.1 Resource control supporting components

This RCF specification in D2 provides a foreground description of controlled node resources, their general behaviour and means to control a combination of these resources. It can be used as a basis for a refined design of resource control service supports such as DENES, and for their implementations. A typical refinement on the current design of DENES, that focuses on controlled packet forwarding resources, would describe means to combine physical packet forwarding resources (I/O ports and associated buffers, bandwidth) and logical packet forwarding resources (I/O queues, classifier tables, forwarding tables, etc), to provide complex resource specifications in the DENES sense i.e. (packet forwarding resource) reflectors, that will implement objects running in DENES engines (entries, traffic managers or traffic controllers). RCF interfaces and resource behaviours defined in D2 can be also used to identify functions of the basic resources that can be modified according to DENES principles and how do proceed, while preserving the overall integrity of the system. In turn, the derivation of DENES principles related to the modification of computational behaviours can stimulate further modifications in next versions of the behaviour model of resources.

7.3.4.1.2 Local and end-to-end admission control with DENES

In the previously presented admission control process scheme, admission control operations are performed during the allocation and the renegotiation of VE resources, in order to identify and provide the amount of resources used by a VE. An admission control operation is performed during the creation of a VE, or for the renegotiation if its resources. Such an admission control system can act as a local client of a DENES engine, to cover both the resource control and monitoring operations. Admission control can be dealt with in a more general sense, which involves resource control operations for several active nodes. These kind of problems are usually referred to as admission control for end-to-end services, and they rely on the use of multiple and “simultaneous” resource control operations on individual nodes involved in the end-to-end service.

One of the main design goals of DENES is to enable the use of DENES engines running on concerned nodes service as the resource control infrastructure for end-to end services, in particular: network control and application services, distributed network management services.

7.3.4.2 Security considerations

7.3.4.2.1 Some potential threats

DENES engines allow prospective clients to download executable codes used in computations associated to controlled resources, in replacement of existing ones. For example, executable codes can be transmitted with a request addressed to a reflector of an input port, as a mean to indicate the procedures to use to handle packets arriving at that port, for the requested service. The replacement of an existing behaviour can be done temporally or permanently. In this perspective, there is a risk of an inappropriate functioning of an input port due to new codes, or even a disorder of the overall packet

forwarding system. Some security check operations can be performed during the development of client programs using e.g. programming languages for active networks with restrictions or verifications related to the secure use of node resources (e.g. access to some memory areas, conditioned invocation of some operations, detection and elimination of infinite loops in some specific conditions, etc.), but these kind of verification can hardly prevent the occurrence of problems during operations, or solve them. Following are some (non exhaustive) examples of security problems that can occur during the operations of a DENES engine:

- Unauthorized modification of resources: physical and simple logical resource, composites resources, and their reflectors. Consistency of the reflection system.
- Unauthorized manipulation of data and state information.
- Unauthorized invocation on reflectors and hazardous operations associated to new code.
- Intentional or unintentional concurrency and resource sharing problems (e.g. famine), especially with clients providing end-to-end network and application services.
- Integrity of resource chains, e.g. chains of pointers or references used for the forwarding of a packet from an input port to one or many output ports within the node.
- Isolation and protection of packet belonging to different traffics: operations of a DENES engine should not allow the violation of measures taken elsewhere to ensure the integrity and the confidentiality of packets belonging to different traffics.

7.3.4.2.2 Protection with the FAIN security framework

We expect the FAIN security framework to apply to resource control systems as well. The current approach of the FAIN security framework is to develop means to protect node infrastructure services from security hazards due to application codes. This can apply to resource control service supports as well, i.e. protect a resource control system from hazards due to its clients, provided that the scope of the security framework is not to much restrictive, e.g. being applied only to EEs or VEs.

7.3.4.3 Role in EEs and Ves

7.3.4.3.1 Alternative choices

Depending on the capabilities and the functions of the node, a DENES engine can be implemented for:

- Resource control operations related to a single EE.
- Resource control operations related to a single VE, that perform resource control operations related to one or several EEs running in the context of that VE.
- Resource control operations related to the overall node, which perform resource control functions related to one or several VEs supported by the node.

The design of DENES does not constraint a specific choice. The only constraint is that resources controlled by a DENES engine and the engine via a single addressing space, must manipulate the controller itself in order to simplify problems related to communications and synchronizations between services supporting the execution of DENES objects, and interactions between controlled resources (their reflectors). Depending on the partition of node resource spaces, a DENES engine can serve the entire node (single address space for all VEs and EEs), a single VE or a single EE. It is also possible to leave the choice of the number of engines per node and their scope to the implementer, but their choices need to be clarified in the related documentation. Using different choices will probably complicate prototype implementations. Thus, this possibility is left to future DENES specifications.

7.3.4.3.2 EE specific DENES engines

In this case, a DENES engine controls resources used by a single EE. A node runs as many DENES

engines as EEs running on the node with resource control requirements. A block of resources is typically reserved for individual EEs during their creation (e.g. through the execution of an admission control service), while resource requests associated to traffics flowing via that EE will be executed by the associated DENES engine, just before the traffic starts. In case of an overload of traffics in that EE, clients can be allowed to use another EE within the same node, provided that it can interpret the same packets. An active node in this case typically presents several traffic manager interfaces to local clients and the outside world (one per EE), each of these traffic manager interfaces offering operations to control several traffic managers and entries.

7.3.4.3.3 VE specific DENES engines

In this second case, a DENES engine controls resources used by one or several EEs running within the context of a single VE. A node runs as many DENES engines as VEs running on that node with resource control requirements. A block of resources can be reserved for individual VEs during their creation, using an admission control mechanism, while resource requests associated to traffics flowing via an EE of a given VE are executed by the DENES engine associated to that VE. In case of an overflow of traffic in a VE, clients can be allowed to use another VE within the same node, but this is rather unlikely (at a first analysis), since a VE is typically used for communications related to a customer or a private network. However, a customer can have many virtual networks, or several customers can share one or several virtual networks, provided that their environments are designed to ensure traffics coming from different customers, and that the business related problems (confidentiality, security, mutual trusts, etc.) can be ensured.

An active node in this case typically presents several traffic manager interfaces to local clients and the outside world (one per VE), each of these traffic manager interfaces offering operations to control several traffic managers and entries.

7.3.4.3.4 DENES engines serving multiple VEs

A DENES engine controls resources used by all the VEs and EEs of the node. A node runs a single DENES engine for all the resource control requirements expressed by applications. A block of resources can be reserved for applicative environments at the installation of the system, other resources being reserved for different system software. As in the first two cases, an admission control mechanism can be used for the creation of the application environment, while the DENES engine is used for resource control associated to incoming traffics. In case of resource overflow, all the incoming requests are constrained to be routed via other nodes. An active node in this case presents a single traffic manager interface to local clients and the outside world, each of these traffic manager interfaces offering operations to control several traffic managers and entries.

7.3.4.4 Relations with demultiplexing systems

Our view of the relations of resource control and DENES with de-multiplexing systems, is that the later one provide means to route both resource control request and traffic flows to the appropriate EEs and VEs. Thus de-multiplexing systems provide a binding between some of the packet forwarding resources (e.g. input ports) and VEs/EEs. As such, they are potentially subject to resource control operations, since, their functioning can have an impact on QoS expressed by applications. A node with multiple DENES engines relies on the proper functioning of de-multiplexing systems for the transmission of requests and client packets flows to the appropriate EEs and VEs.

7.3.5 Summary of the DENES system design - next steps

This section presented the principles of DENES, a resource control system that will be used to enable new application and network services and network management services, especially those that rely on end-to-end QoS constraints. The presentation also focused on DENES engine that ensures strictly reliable communications, and the relations of a DENES engines with other FAIN node services and frameworks. This presentation opens the door to detailed DENES specifications with interfaces, prototype implementations, and further studies for the extension of its functionalities.

8 EXECUTION ENVIRONMENTS

In real world application of active networks, an important aspect of usability is the degree of flexibility offered by an active network node. The degree of flexibility on the one hand depends on the manageability (cf. WP4, Deliverable D5) of a node but on the other hands it depends on the execution environments (EE) since the EEs provide the run-time environments where foreign code can be installed and run. The installation and execution of foreign code must be achievable neither resulting in a compromised node nor in an interruption of service of an active node.

In FAIN, research on execution environments has revolved around a specific type of EE that manifests the property of flexibility and in particular the composability and extensibility as argued in section 2.1. Two instances of this type of EE has been designed and implemented. One instance, presented in section 8.1, is based on JAVA/CORBA technology, which runs in the so-called user space of an active network node, whereas the other, presented in section 8.2, is a high-performance execution environment which runs in the kernel space of a Linux operating system. Finally, another type of control EE has been developed, presented in section 8.3, for deploying control protocols, which makes use of SNAP EE and SNMP for controlling system resources.

8.1 JAVA EE

Since the active node management level is implemented in JAVA it requires a JAVA execution environment for running management components. The JAVA execution environment can also be used for running services implemented in JAVA. The JAVA components can be re-used as wrappers for implementations in a non-JAVA environment. This is done for example for demultiplexing, security, and traffic control.

The implementation of the management level uses CORBA for communication between objects. Thus it provides strong support for CORBA ports, which can easily be added and removed by component implementations. Also the control of the access to CORBA ports and the later usage by clients is handled by the JAVA management environment and allows an easy connection to the security component. This functionality is also available to general service components.

The main classes are:

- **BasicComponent**, implementing interface *iComponentInitial*
  - get the unique ID of the component,
  - get descriptions of all offered ports by this component,
  - get access to a specific port.
- **ConfigurableComponent**, derived from BasicComponent, implementing interface *iConfiguration*
  - get, set, and change the component's properties,
  - connect and disconnect ports of the component to other ports,
  - suspend and resume the execution of the component.
- **ComponentManager**, derived from ConfigurableComponent, implementing interface *iComponentManager*
  - create and delete component instances,
  - activate and deactivate component instances,
  - get a list of component instances.
- **VirtualEnvironmentManager**, derived from ComponentManager, implementing interface *iVirtualEnvironmnetManager*

  - create and delete VE instances,
  - activate and deactivate VE instances,
  - handle creation, activation, deactivation, and deletion of resources needed for a VE instance,
  - get a virtual environment by specifying the VE ID.
- **ExecutionEnvironmentManager**, derived from ComponentManager
  - create and delete EE instances,
  - activate and deactivate EE instances,
  - map EE instances to operating system resources.
- **VirtualEnvironment**, derived from ConfigurableComponent, implementing *iTemplateManager*
  - install and uninstall templates (i.e. service managers),
  - hold references to attached resources,
  - dispatching installation and uninstallation to appropriate execution environments.
- **ExecutionEnvironment** (JAVA), derived from ConfigurableComponent, implementing *iTemplateManager*
  - install and uninstall templates (i.e. service managers) for JAVA services,
  - use JAVA class loader for installation.

8.2 High Performance EE

The high-performance EE is required for the installation of code that processes data at line speed. Example applications could be transcoding of multimedia streams, extended, adaptive packet filtering for firewall issues, distributed Web Cache functionality, load balancing etc.

Regarding performance, one of the parameters of an EE is where the EE is instantiated. In modern operating systems (OS), a clear demarcation line is drawn between the so-called kernel and the so-called user space. Kernel space is where the basic functionality of an operating system is located like process, memory and IO management while user space is most often seen as the address space where user applications like word processing etc. are run.

Most often supported by the underlying hardware, user space and kernel space differ in privileges and protection boundaries. For example, kernel space usually got assigned privileges that are required to manage and operate a node while user space application run in a protected environment where code is not allowed to cross boundaries imposed by the operating system, i.e. they run as a task inside an own protection domain.

Crossing protection domains is a time consuming process. Switching privileges requires even more processing and thus costs more time. Execution environments may be placed in any of the two principal address spaces, i.e. they may reside either in the user space or in the kernel space if the operating system provides this differentiation at all and does not run as a so-called single address space operating system. While running an EE in user space provides several advantages regarding the ease of programming and pre-available operating system support for strong protection, stability and dependability of the node – a user process should not be able to compromise a complete node, usually --, the location of EEs in user space imposes more overhead and thus results in a less performant node. A very well known problem is found in the crossing from kernel space to user space protection domains in Linux. Latest results in measuring these costs prove this problem.

One way to address the performance issues of a node could be to run active applications in traditional kernel space. So, the switching of protection domains and privileges can be avoided. This, of course

immediately leads to the concerns regarding erroneous codes, security and stability, i.e. node safety in general, as active applications running in traditional kernel space would have the power as any other kernel component.

Another way to address performance issues while trying to keep protection wherever possible is addressed by the PromethOS architecture. In PromethOS, the kernel space EEs are designed as their own protection domains. Based on memory management capabilities, an EE in kernel space should be able to run in its own virtual address space. Thus foreign code running inside of it is not allowed to cross the boundaries.

8.2.1 Model

In FAIN, the high-performance EE is modelled according to the Mach Task model, which is similar to the one used in UNIX. A Mach task, as defined by the Open Software Foundation, is a "container that holds a set of threads. More importantly, it contains those elements that the threads need to execute, namely a port name space and a virtual address space.". The model of the FAIN high-performance EE extends this model by that it allocates resources to a PromethOS-EE based on the specifications provided for a Virtual Environment (VE).

Even though, the user space processes of a modern operating system follow the Mach Task model most often, and thus the complete PromethOS node could be modelled by this principle, we restrict ourselves to the PromethOS kernel space aspects. Thus, when referring to a PromethOS node, we implicitly refer to the kernel space NodeOS aspects and the PromethOS-EE as well as the PromethOS plugins.

Internally to a PromethOS-EE, the plugins are organized as directed graphs. A graph may contain forks and branches. Different to many other approaches, these bifurcated branches may be concatenated later.

A graph of plugins provides the processing path of arriving data packets. Every plugin along the path gets the chance to process the packet. PromethOS provides the mechanisms to connect the plugins in a processing path, and offers the methods for inter-communication between a pair of plugins. However, the specification of which plugins need to be interconnected is required to be provided to PromethOS by the ASP component.

Resources controlled for the high-performance EE are defined as CPU cycles, memory consumption and IO bandwidth. Similar to Mach, where the containers are managed and controlled by a task manager, in PromethOS the EEs are controlled and managed internally by the PromethOS management component. So, PromethOS provides the framework that is used to create, set-up and control a high-performance EE.

8.2.2 PromethOS v1 Requirements

PromethOS needs to offer the following properties:

- Dynamical loading and disposal of plugins into the operating system kernel at run-time. Plugins are code modules, which implement a specific router functionality. For example, a router plugin might implement encryption functionality.
- 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 plugins 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.
- A consistent and simple interface must be provided such that the plugins may be easily programmed. A plugin must react to several signals. In PromethOS, the plugins must react to a set of such signals. By a unified set, interoperability of plugins is provided.

- Efficient mapping of specified flows and the possibility of binding flows to specific plugin instances is 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 need a filter specification according to the aggregate level: a single flow requires the specification of all parameters.
- High throughput along the whole data path. High throughput is achieved partly by the implementation residing fully in the kernel. So, expensive context switches may be omitted. To another degree, efficient packet classification is required, too. Packet classification allows the binding of flows to plugins.

8.2.3 Netfilter Architecture

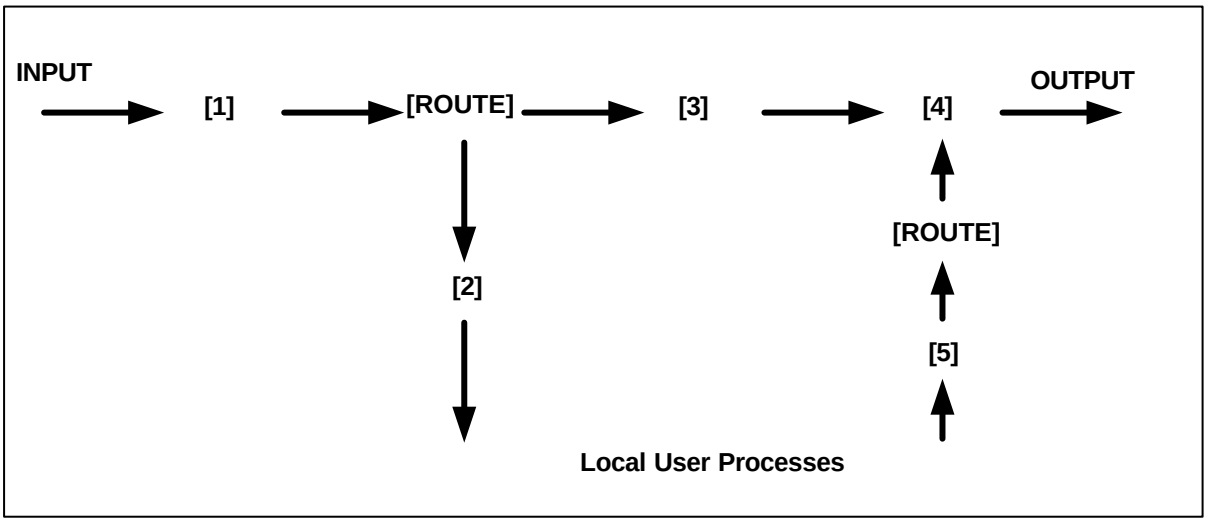


Figure 8-1: Netfilter Architecture for IPv4

PromethOS is based on the Linux Netfilter framework for packet mangling. As such, it is integrated into the Netfilter framework. PromethOS is loaded as a Linux kernel module at run-time. It registers at the different hooks of the Netfilter framework such that it may receive packets from Netfilter framework, and feed them back into Netfilter without violating the structure of the Linux network stack. The decision for this implementation was taken since the Netfilter framework provides a perfect environment for extensibility and has proven to be fairly stable.

Netfilter, developed and implemented by Rusty Russel, 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 enquire for user space. Figure 8-1 provides an overview of the Netfilter framework for IPv4.

8.2.4 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, Netfilter needs a Netfilter-extension that enables the kernel space packet traversal path to install plugins at run-time without the need to be specifically built for one kernel only. This Netfilter-extension, as provided by PromethOS, provides the mechanisms to load plugins and bind them to specific flows. An overview of Netfilter together with PromethOS is provided in Figure 8-2.

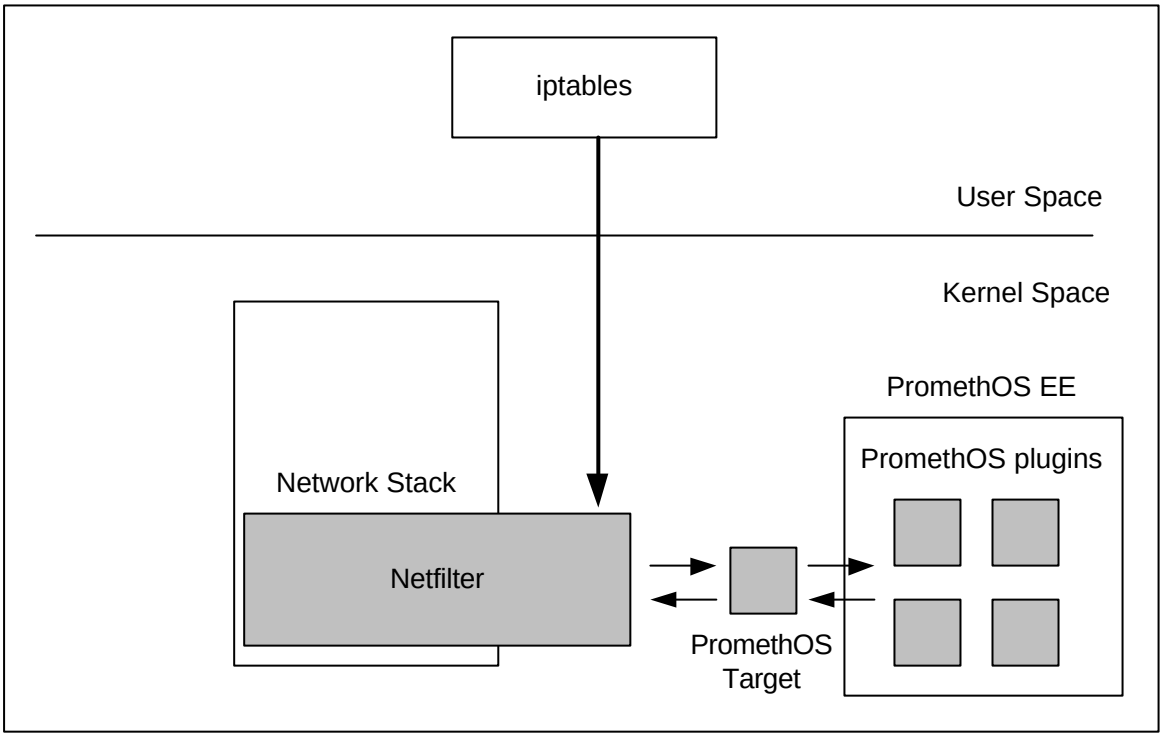


Figure 8-2: Netfilter and PromethOS

The following functionality needed to be implemented:

- A new Netfilter table into which filter expressions may be specified. These filters may bind plugins bound to flows.
- A new Netfilter target[1] must be implemented. By this target, plugins are managed and controlled at run-time. This target dispatches packets to the appropriate plugin instances. So, it classifies packets for flows.
- iptables as the user space tool must be extended such that the new arguments can be passed to the new Netfilter target.
- Control functionality should be provided to query statistical information.

8.2.5 Implementation

In this section, we describe PromethOS v1 as it is expected to be available for Milestone M3 in June 2002.

8.2.5.1 PromethOS Netfilter-Table

To easily separate PromethOS flows from normal filters, and to allow the filters to be hooked to every of the available hooks, a new Netfilter-table, PromethOS, is implemented. This table need to register

[1] The Netfilter-target provides the PromethOS framework. The PromethOS framework controls packet dispatching and plugin installation.

during load time at the Netfilter framework. Obviously, during removal, it must deregister as well.

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 8-3 provides an overview with the example of an IPv4 protocol-hook configuration.

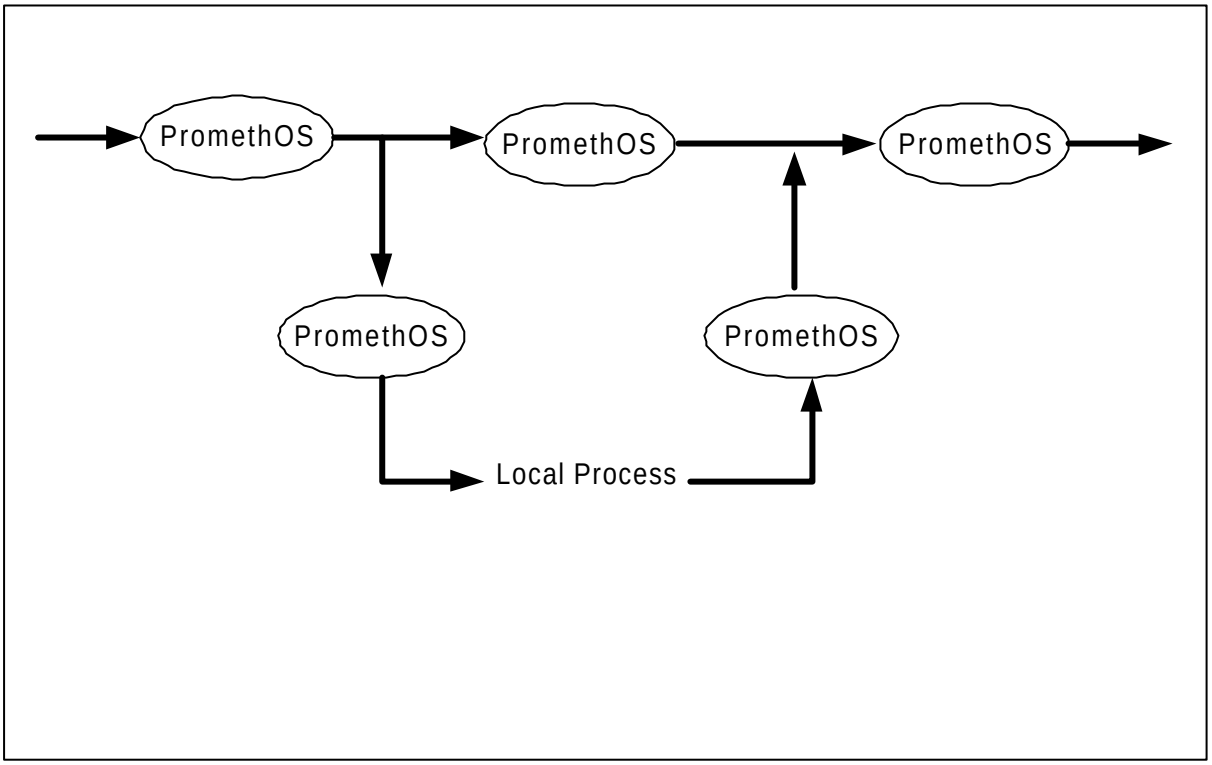


Figure 8-3: PromethOS Netfilter-Table Hooks

8.2.5.2 PromethOS Netfilter-Target

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. A control functionality is provided that keeps track of the loaded plugins and their instances.

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.

8.2.6 PromethOS v1.0

Figure 8-4 presents an overview of a PromethOS node. We identify the following entities:

- PromethOS: framework management component. It provides the NodeOS functionality and resource control and framework internal management functionality.

- Node Management Component: Comprises all node management components as required for managing a PromethOS node. The node management component is outside of the scope of implementation work carried out by the PromethOS team. However, the interfaces allow for an integration of both, the node management component and PromethOS for the final milestone of FAIN in year three.
- Demultiplexing and Multiplexing: These components provide the demultiplexing according to configurable settings. They may be re-configured at run-time.

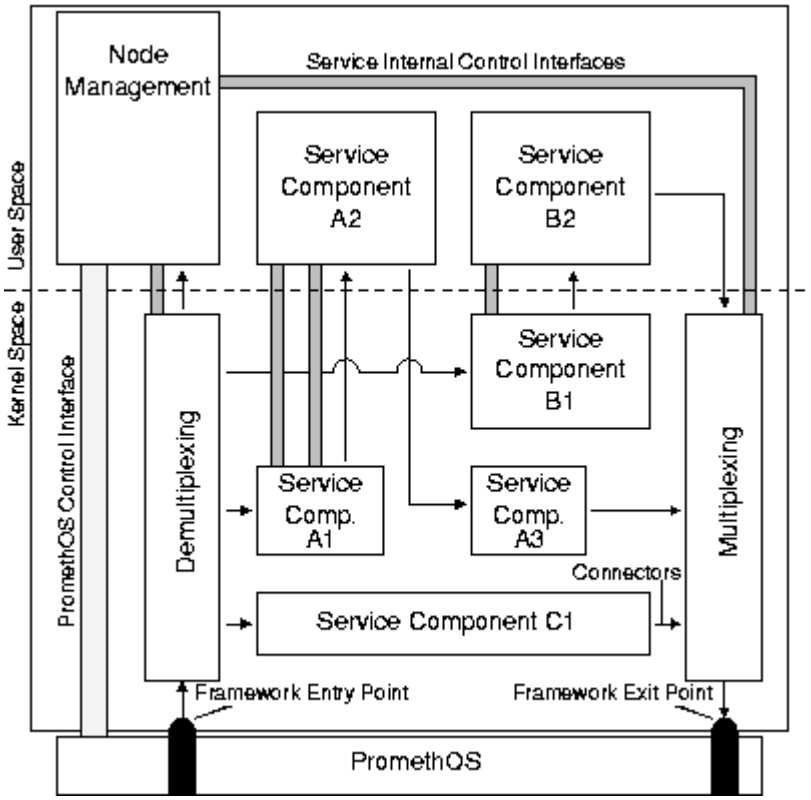


Figure 8-4: PromethOS Node

- Control Interfaces: One interface between the node management component and PromethOS, and several service internal control interfaces between service components in kernel space and user space. Demultiplexing and multiplexing are provided as components. So, they may offer control interfaces to the node management component.
- PromethOS framework entry and exit points: They provide the interfaces, which are connected to the Netfilter framework.
- Service Components: Components used for the composition of services.

Hereafter, we describe each component in more detail.

8.2.6.1 PromethOS Framework

The PromethOS framework is internally managed by the PromethOS component as depicted in Figure 8-4. It provides the NodeOS functionality of a PromethOS node. This component is responsible for the control and enforcement of resource consumption[2] and limits respectively. Further, the communication infrastructure is provided by this component as well. As depicted in Figure 8-4, this component is attached to the network stack of Linux.

[2] Resource Control will not be available for the first release of PromethOS.

8.2.6.2 Node Management Component

The node management component is responsible for managing PromethOS. It installs code, creates virtual environments and binds code to VE for resource control. Further, it is responsible for node management issues. Among others, the node management component comprises the resource control framework and node local service deployment. Further, it specifies the interconnection of components to PromethOS such that this can establish the connections.

8.2.6.3 Demultiplexing and Multiplexing

These components provide the demultiplexing/dispatching and multiplexing functionality. They are configurable at runtime. The demultiplexing component dispatches packets to the configured services.

8.2.6.4 Control Interfaces

Control interfaces exist between a kernel component and a user space component. Kernel space components may export them optionally. The kernel components involved in a user service as well as PromethOS itself offer them via the /proc-Filesystem of Linux. Management components (service internal as well the node management component) may attach to these interfaces.

The interfaces are documented in the internal report R10 of WP3.

8.2.6.5 PromethOS Framework Entry and Exit Points

They provide the interfaces that are connected to the Netfilter framework. They are implemented as Netfilter hooks.

8.2.6.6 Service Components

Service components are kernel space and user space components used for a customer service. PromethOS handles them in a uniform way for resource control and communication. Kernel space components are managed as PromethOS plugins. The process model of Unix handles user space components.

8.2.6.7 PromethOS Plugins

An overview of a PromethOS plugin at run-time is provided in Figure 8-5.

A PromethOS plugin is confined by several input and several output channels, and by a control interface which is optional. The plugin must explicitly register for such a control interface while at least one of the input and output channels are a fix requirement. The control interface may be used to configure and re-configure the plugin at run-time, or for gathering statistical information depending on the functionality the plugin offers.

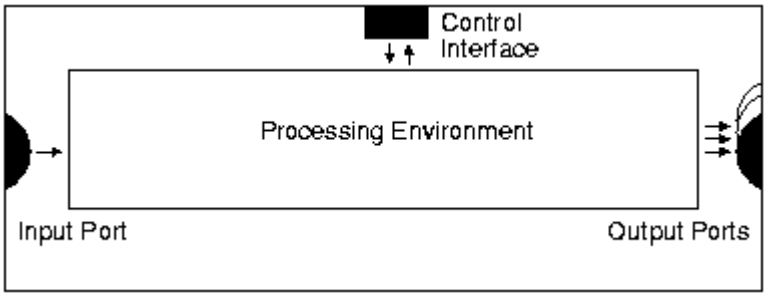


Figure 8-5: PromethOS Plugin

Loaded plugins are interconnected such that they form a directed graph of modules. This graph defines the sequence of execution. However, the design of PromethOS allows this graph to be forked. As such,

the graph forms a tree. The PromethOS communication infrastructure handles the packet dispatching according to an internal signalling protocol. This signalling protocol is used such that the plugin can instruct the communication infrastructure to which channel packets must be forwarded.

8.2.6.8 PromethOS Execution Environments

PromethOS provides a kernel space execution environment (PromethOS-EE). A PromethOS-EE is created by the interconnection of PromethOS plugins.

A PromethOS EE follows the model of a UNIX process or a Mach task. It serves as a container into which different plugins can be dynamically loaded and then run. Following this model, the PromethOS EE defines the boundaries for code running inside of such an EE. PromethOS EEs form the entity which are assigned to customers, i.e. to Virtual Environments (VE) in line with the FAIN node architecture. Several PromethOS EEs may be assigned to a single VE, but not the other way round (one EE assigned to several VEs).

The PromethOS EE is instantiated by ways of Netfilter tables: one table per EE instance. However, while Netfilter currently supports only statically pre-configured tables, PromethOS extends this framework with mechanisms to instantiate an unlimited number of PromethOS EEs at run-time. However, the multiple EE instance facility may not be ready for M3.

8.2.6.9 Customers and Virtual Environments

PromethOS must control resources assigned to customers such that resources are only consumed as contracted on the one hand and on the other hand such that resources are available for different customers. For this reason, FAIN introduced the Virtual Environment (VE).

A VE is similar to a domain as defined by D2. The virtual environment as used on a PromethOS node identifies a customer, and, thus, the specification of resources for a set of instantiated services that belong to the customer. Assigning instantiated services to a VE allows for sharing of resources.

In PromethOS, Virtual Environments (VE) are built as a set of service components, i.e. components that may reside either in user or kernel space. A VE is an abstract identifier for aggregated resource accounting and control. It might be associated with a customer on behalf of which resources are charged. For PromethOS the clear definition of resources that might be consumed by service components bound to a VE is required. Service components are bound to VEs for resource control issues.

8.2.7 Interfaces

In PromethOS, two types of interfaces are required: Interface type number one that is used to create the PromethOS EE instances and populate them with PromethOS plugins, and interface type number two that is used for run-time control of PromethOS and the PromethOS plugins. The former is implemented for PromethOS v1 by using an extended version of the Linux iptables user space application, while the latter is implemented by the well known /proc Filesystem of Linux.

8.2.7.1 Run-time Control Interfaces

The PromethOS framework creates a management file in the /proc file system of Linux. All control interfaces related to PromethOS, i.e. PromethOS framework control as well as PromethOS plugin control, are located below /proc/promethos. Since PromethOS will be extended with resource control in the final project year, a differentiation is made between network related and node control related issues. So, at M3, the control interface offered for run-time control is accessed via reading and writing the /proc file system entry identified as /proc/promethos/net/management.

The interface /proc/promethos/net/management is used to carry out system control functions. For the final milestone, it is assumed to instruct PromethOS regarding network-oriented resource control by this interface.

Instantiated plugins offer a control interface by using the same methods of Linux. If requested at plugin load time, a plugin control interface can be accessed via /proc/promethos/net/management #<plugininstancenumber>, where <plugininstancenumber> is the number assigned to the plugin by PromethOS at instantiation time. Using this interface, the plugin can be configured at run-time. The exact semantics and syntax of the plugin control interface is not restricted in any way by PromethOS. Thus, the PromethOS plugin programmer is free to define its own protocol for performing service internal control. It is assumed that user space service components carry out service internal control by attaching itself to the appropriate interface and do a request-oriented command issuing, i.e. the plugin react only on request issued by the user space application.

8.2.7.2 iptables

Iptables is the user space tool that directs the framework to instantiate PromethOS EEs and to load PromethOS plugins. Node-unique instance numbers identifies Plugins. To achieve this functionality even on a multiprocessor system without huge overhead, the /proc-Filesystem is used as well. An instance counter delivers unique numbers to the user space (/proc/promethos/net/instance).

The User Space tool 'iptables' has been extended. New flags for the PromethOS framework must be passed at load-time. Help texts are provided as well. This is achieved by including a new shared library to the iptables tool.

8.2.8 Outlook, Further Work And Conclusion

8.2.8.1 Expected Achievement until M4

For milestone M4 at the beginning of June 2002, PromethOS will provide the mechanisms to differ among several users and allow several instantiations of several plugins that may be bound to different flows each. However, resource control carried out by PromethOS will not be available for M4.

8.2.8.2 Outlook and Further Work

One of the clear requirements for the FAIN active network node is the integration of PromethOS into the management framework such that PromethOS can be managed by the same approaches like the remainder of the node is managed. For the release of PromethOS at the end of the project, resource control for PromethOS EEs is required. Aspects of high-performance must be investigated and the trade-off of flexibility and node reliability must be evaluated. Further plugins are required to provide functionality that can be installed on a FAIN active network node in the high-performance execution environment. And finally, the PromethOS approach must be evaluated and compared to other approaches in the area of active networking.

8.2.8.3 Conclusion

In this section, we described the architecture of a PromethOS node in its first release. We introduced the PromethOS plugin, the PromethOS-EE and the explained the functionality of a VE as it will be handled by PromethOS.

8.3 SNMP Activator: A Resource Control Facility for Network Resource Allocation

8.3.1 Introduction

8.3.1.1 The fundamentals: Applying active network technology to network management

The is primarily an activity between Network Infrastructure Providers (NIP) in response to some network management signal. The signal is discussed more substantially in WP4 documents. For now, it is sufficient to say that NIPs have to allocate resources across the network to allow virtual environments to attempt to load execution environments.

It is worthwhile reviewing some of the fundamentals of the application of active network technology to network management.

8.3.1.2 Traditional Networks

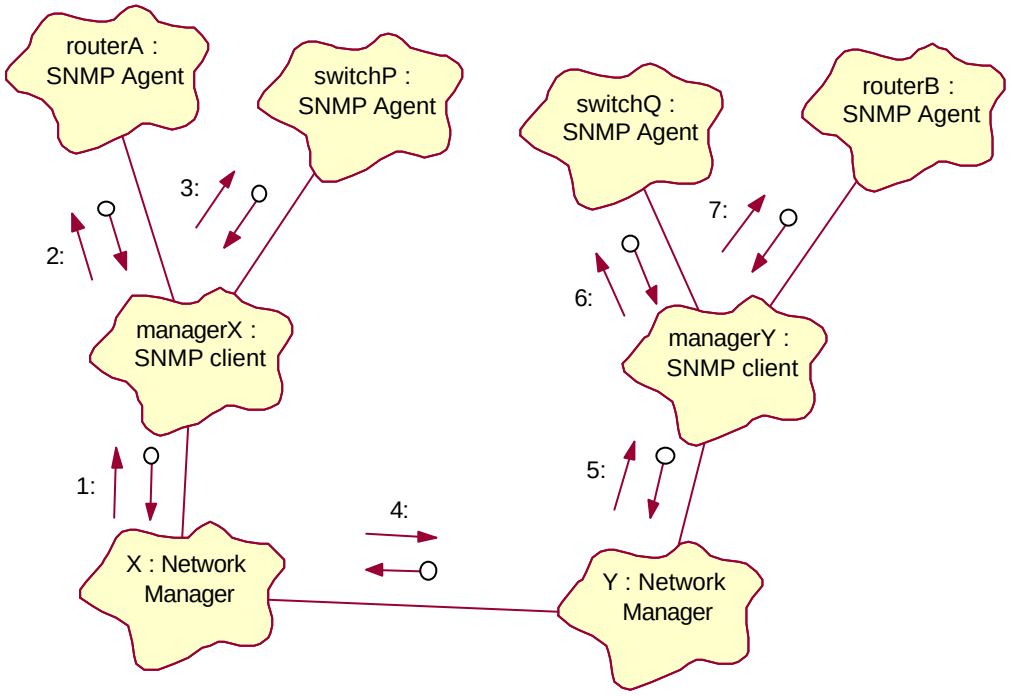


Figure 8-6: Conventional Polling NM

Traditional networks rely upon:

- *a priori* knowledge of network device location, and
- polling to determine what traffic has arrived at a router.

(Conventional routers do not usually even provide this information and all the network management can do is monitor routes and ARP information.)

Diagrammatically a simple client-server model using northbound telecom interfaces can represent this. Figure 8-6 illustrates this with the usual added difficulty that the control plane of edge networks is not usually routable across the public Internet (i.e. The routers and switches are located in a private network: 10.0/8 etc.).

In this scenario, a provider edge network manager X uses an SNMP client in his private control network to manage router A and switch P. He then communicates (by telephone, secure e-mail) a set of instructions to his fellow network manager Y and asks him to program the devices in other edge network.[3]

The system is secure, but is based on human trust relationships, which are intransitive (they do not commute to institutions) and have a temporal dimension (largely based on "tit for tat" exchanges.). The design can be improved by allowing X to use SSH to login to the private network of Y and use the SNMP client directly but the cost is leaking information of the construction of the network-to-network manager whose integrity may be compromised.

The greatest cost to the network manager is location of the routers. He would use a utility such as traceroute to locate the routers he wishes to configure, locate his counterpart, organise some mutual exchange of trust and finally implement the reconfiguration.

In the current Internet, this is not so impractical. Routers do not have much information to use for feedback control and what they have is either so well-managed by a mechanical protocol (RIP or BGP or OSPF for routing) that network managers rarely need to intercede or is useful only for static policies such as choosing a network provider. (For example, a network interface may suffer a large load, buy another one and use it 24/7, i.e. All the time.)

Finally, we should note that in the conventional Internet routers can do very little. Most routers will support DHCP, packet filtering and network address translation, RIP, IPSec, GRE and perhaps DiffServ, but there are very few embedded system routers that implement genuinely novel data transcoding methods.

8.3.1.3 Active Networks

8.3.1.3.1 Location Discovery: Event Channel

The single most constraining feature of the operation of the conventional network management model is the need for *a priori* knowledge of the location of routers.

Using a packet interceptor, it is possible to generate a signal to the router that a particular packet has arrived. This feature of modern packet filters was presented at the first project review when ABLE was demonstrated. That network management application demonstrated an initiator being accepted for an IPSec tunnel. In the second project review, the application was developed and an IPSec acceptor was created and, on arrival at the remote network, an initiator was created there.

In architectural terms, the packet interceptor pushes an event onto an event channel to its network manager. It cannot be stressed too much how useful this feature of packet filters will be for network management. A packet now has the ability to arrive at any router in the Internet and raise an alert with its originator's service provider. This will allow network users to quickly attain the level of network service they usually expect.

This feature will be coupled with highly programmable routers to provide a means to rapidly establish ad-hoc networks for Internet users. Networks that will comprise:

- traffic-conditioned tunnels to favoured content providers
- encrypted tunnels to secure content providers
- "magnetic" cache servers that migrate towards network users containing their most recently used information.

[3] Unfortunately, these diagrams were produced before the FAIN reference testbed diagram was added. managerX should be EMS1A, managerY should be EMS2A; routerA and routerB should be AN1 and AN2 respectively.

- Network address translation facilities to allow roaming users to retain their Internet identity in different administrative domains.

Such ad-hoc networks would be an excellent solution to many of the problems presented by Mobile IP systems.

8.3.1.3.2 Router and Network Device Programmability

Internet router functions have improved phenomenally. A Linux or BSD router can provide packet counting and logging, a variety of traffic conditioning packet filters; it can adapt packet processing and routing according to the sub-protocols defined in the IP packet.

A PromethOS/Netfilter-enabled router can implement sophisticated stateful packet filter methods and can route FTP packets via a different network that telnet or SSH.

And, of course, a Linux or BSD router will support all of the functions available to a conventional embedded system router.

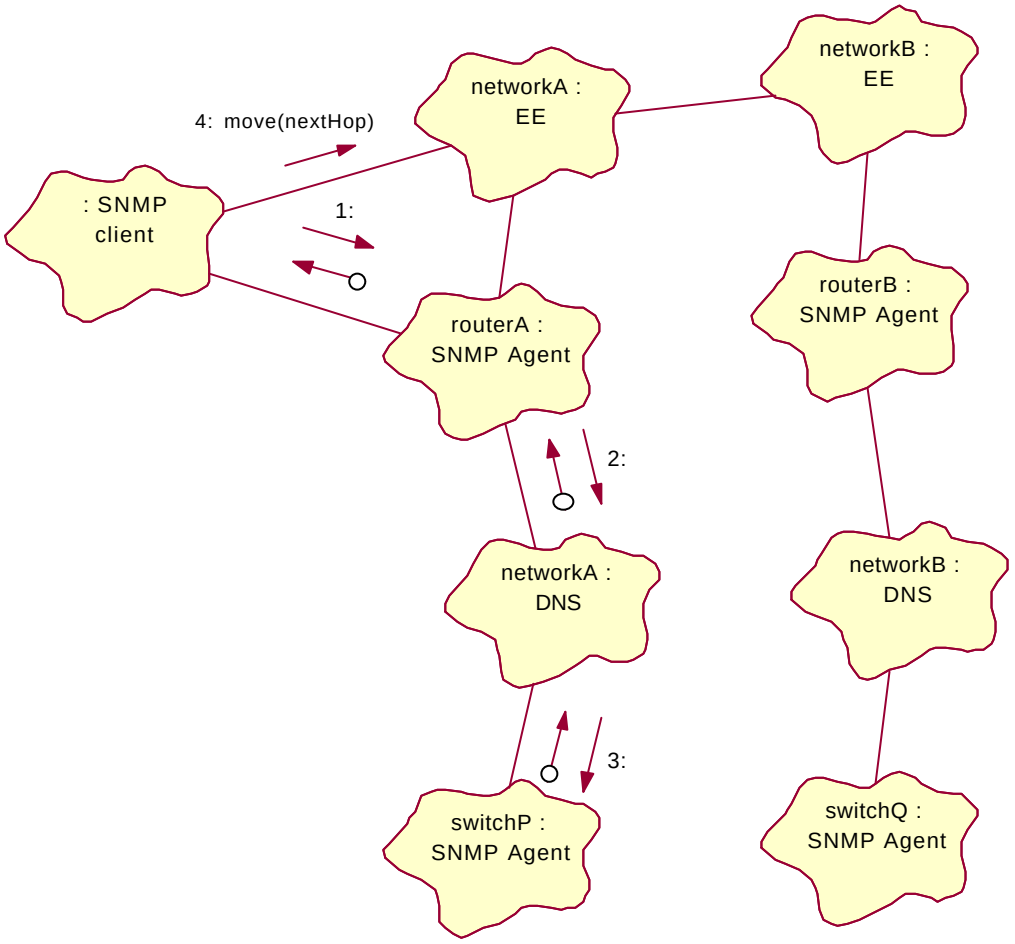


Figure 8-7: Active SNMP Client: Epoch 1

Most importantly, a Linux or BSD router will also offer a variety of user-space daemons that can be used for monitoring and configuration management.

Programmable private ad-hoc networks are now a reality, what is needed is a means of rapidly reconfiguring the network.

Rapid reconfiguration of the network can be achieved by using active packet technology. In Figure 8-7: Active SNMP Client: Epoch 1, we see an SNMP client that is initially running in an Execution Environment in network A and uses this to interrogate a router, it also knows the location of the Domain Naming Service for the private control network that the router is part of; it uses this to locate switch P. Once it has completed issuing its instructions in networkA e.g. updating the routing table it instructs its EE to move it to the next hop: an execution environment in network B.

In the second epoch of the SNMP client's existence, Figure 8-8: Active SNMP Client: Epoch 2, we see the same process carried out in network B. This will be repeated until the packet containing the SNMP client reaches its final destination network.

8.3.1.4 Management and Control Plane Separation

A clear description of the nature and the responsibilities of the management and control plane will ease the explanation of the interaction between the SNMP activator and the other control components, which have been developed in FAIN. The management plane contains management functions that are ready to receive network status information from the network entities. The management plane is responsible for how resources should be allocated among different network devices, and it provides services such as *configuration management* which specifies the management of physical resources and how they are included into the configured network; *resource management* which manages the physical construction of the network; *performance management* which ensures the network is tuned to meet its performance requirements in terms of user perceived QoS; *access and security management* which manages the access rights to the network facilities and network management entities etc.

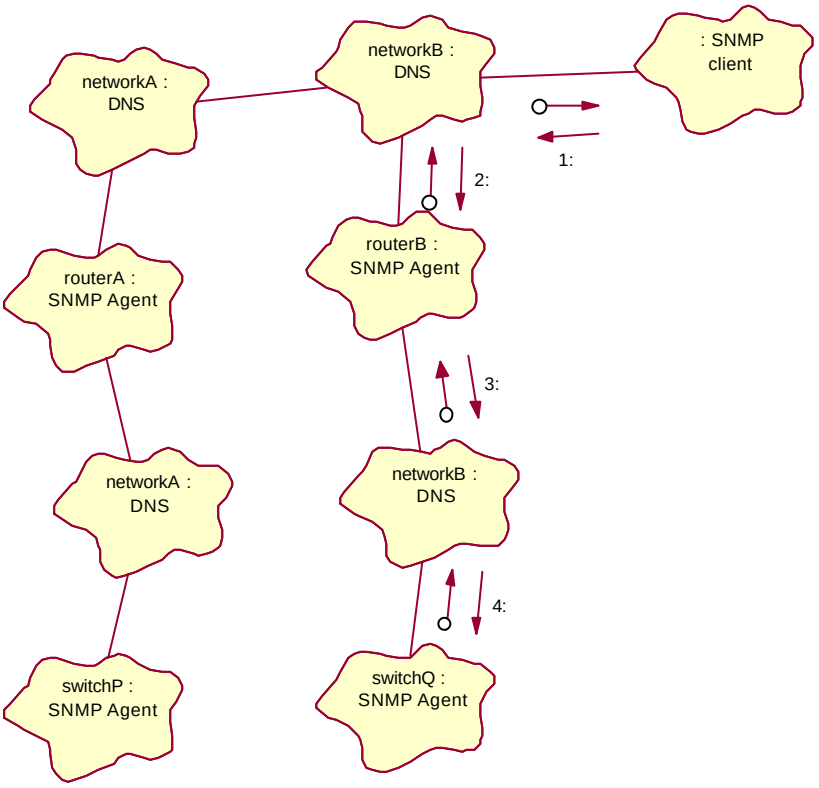


Figure 8-8: Active SNMP Client: Epoch 2

On the other hand, the control plane contains elements such as mechanisms, signalling protocols and algorithms to control many network devices and to provide the logical task of *allocating network resources* among various application flows. For instance, via the control plane a network service

provider can dynamically create, terminate or modify a network service on-demand and to control the usage of network resources.

The management plane and control plane are clearly distinguished in many standards: for instance in ISO-RM-ODP (Open Distributed Processing Reference Model) - the managers collectively form a management agent, whereas each controller is a performative agent that acts upon the decision of the managers.

8.3.1.5 SNMP Activator: A Control Entity

This chapter describes a network management mechanism that is one of the RCF mechanisms for *controlling* the node management resources using SNAP and SNMP across a network. The discussion of the RCF so far has described resources to be controlled within the active node. This mechanism is designed to control the resources around the active node: routers and switches that have a legacy SNMP management system. It is proposed as an entity on the control plane rather than on the management plane to ease management. Together with other control components of the FAIN active node such as the VE management system (which controls VEs and activation of EEs) and PromethOS (which controls the NodeOS plug-ins) it forms a consolidated RCF, which is the component of the FAIN active node, that makes it a significant technical innovation.

The basic interfaces offered within the RCF support the allocation and monitoring of logical and physical resources. The VEs and applications will be interested in the allocation of the necessary resources, while a network management system will use these RCF interfaces to allocate resources using higher-level policies and to monitor the resource usage for accounting and performance management. The RCF mechanism described in this section is able to obtain network resources from any SNMP-enabled network devices that a VE has authority to manage. It uses active packets to implement finite state machines that program a series of SNMP-enabled network devices in a synchronised manner. Most usefully, the state machine would provide a means for rollback: should any request for a network resource fail, and then the fulfilled requests made earlier are released. Using this 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 SNAP programming language to implement the finite state machines. It offers facilities to issue SNMP commands that can be applied to network devices to:

- set and change their current operational configuration - SNMP SET
- get the status - SNMP GET
- set traps to report changes in state - SNMP SET TRAP

It will also be possible to issue an instruction to any active extensions available in (or around) the active node. This will be used to demonstrate the loading of mobile software agents into a Java Virtual Machine near the active node. These mobile software agents will be used for monitoring network conditions and reporting directly to any virtual environments or two other management systems.

(It is also hoped that the extension of Grasshopper by IKV for the FAIN project can be exploited by the SNAP system. The Grasshopper extension allows agents to be transported using ANEP packets.)

Security will be provided for by the standard mechanism used for SNMP: username, password and community. SNAP packets will be transmitted in cleartext, but the authority to action the SNMP commands will be an active extension provided by the virtual environment within the active node. Mobile agents will be loaded in a similar manner while the virtual environment will be given the authority to load them. The mechanism in this latter case will be that available within the Grasshopper agency.

8.3.1.6 ISO-RM-ODP

As mentioned in the previous sections, the concept of the separation between the management and the control plane in FAIN follows the approach of many standards such as ISO-RM-ODP. RM-ODP uses

an object modelling approach to describe distributed systems. Two structuring approaches are used to simplify the problems of design in large complex systems. Five 'viewpoints' provide different ways of describing the system. Each viewpoint is associated with a language, which can be used to describe systems from that viewpoint. The five viewpoints described by RM-ODP are:

1. The *enterprise* viewpoint, which examines the system and its environment in the context of the business requirements on the system, its purpose, scope and policies. It deals with aspects of the enterprise such as its organizational structure, which affect the system.

2. The *information* viewpoint, which focuses on the information in the system. How the information is structured, how it changes, information flows, and the logical divisions between independent functions within the system are all dealt with in the information viewpoint.

3. The *computational* viewpoint, which focuses on functional decomposition of the system into objects, which interact at interfaces.

4. The *engineering* viewpoint, which focuses on how distributed interaction between system objects is supported.

5. The *technology* viewpoint, which concentrates on the individual hardware and software components, which make up the system.

Our SNMP SNAP approach makes several references to the ISO-RM-ODP approach. MIB is used on the information plane to provide the information of network devices; on the computational plane, the protocol is the OMG draft Resource Access Decision Facility; on the technology plane the combined technology of SNMP agents and SNAP packets is used to provide the access to the network resources.

8.3.2 SNMP Activator System Approach

8.3.2.1 Interceptor Paradigm

Active network management is the application area for this system. Active networking is an interceptor paradigm. It is difficult to develop applications that rely upon intercepting data packets because the interceptor must decode the data packet and its intention understood.

8.3.2.1.1 Transcoding and Active Management as examples

Transcoding is a good application for interceptors: it is transparent to the end users. Data can be compressed or encrypted without the users' knowledge.

Active network management is also a good application fits within the interceptor paradigm - it modifies the network to accommodate new data flows. As data arrives on the network, the active network management system generates a network event which is detected and code is injected into the network with this new data. There is no need for a separate control channel. The new control information will manage the new data and precedes it as it traverses the network. Managing network state is difficult, because it is not known how another part of the network is configured without actually visiting it. If the control information is sequence of instructions that are dependent upon the network state, then the program should follow a sequence that will be correct for the operational state of the network.

8.3.2.2 ABLE: Active Networking Out-of-Band

This can be seen in the ABLE platform for network management: Figure 8-9: ABLE-An Example of the Interceptor Paradigm. The ABLE platform used a router's packet filtering capabilities to supply ANEP UDP packets that contained a Java class to the system component "The Activator". The Activator reconstructed the Java class from the packet stream and forked itself. Its child then performed an exec() to launch a Java virtual machine that could run the Java class intercepted it.

ABLE proved to be deficient as an active networking system. It is useful for loading network monitors and managers into routers (or nearby management stations) wherever a customer data flow appeared in the network. It is not suitable for managing the data flow as it progresses through the network. It is, in effect, an out-of-band management facility. It did provide a means to locate flow managers more effectively, but it did not provide a means to locate the flow itself.

This is a problem. A data flow is most unstable when it is first established. The network has to adapt to the load it presents and consequently the nodes through which the flow passes are most likely to change when the flow first presents itself to the network.

What is needed for effective network management is an in-band management capability. Ideally, 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 the SNAP packet will use 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.

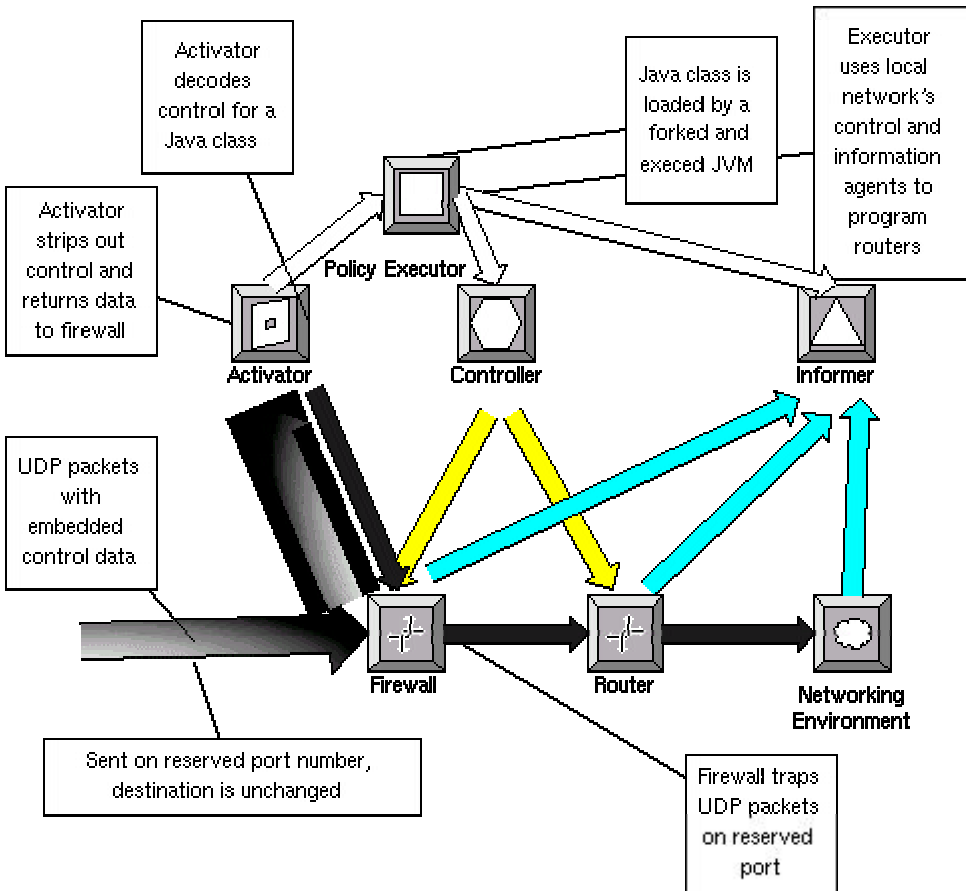


Figure 8-9: ABLE-An Example of the Interceptor Paradigm

In both cases, the data flow will be asymmetric; in the former case, it will require a large 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 it source with the sink of the data flow. The active packet can revise and choose how the source and sink impedances are matched.

8.3.2.3 Introduction to the SNAP Active Packets

SNAP (Safe Network with Active Packets) is a programming language that provides active packets. As a packet traverses the network it can perform computations and add and remove data to a stack within the packet. This is a genuinely active mode of operation. Because the SNAP programming language is a simple assembly language, 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.

The SNAP packets are essentially UDP packets, which are embedded with assembly codes, thus they carry both information and commands. The SNAP packets are executed on stack-based bytecode virtual machines. A SNAP program consists of a sequence of bytecode instructions (pop, forw, push etc), a stack and a heap. The stack keeps smaller data i.e. integers, addresses (which points to the heap values); the heap keeps larger data i.e. byte arrays, tuples (an array of smaller value). SNAP is simple, light-weight, efficient (as SNAP provides only the basic operators and control flow, which leads to a light-weight interpreter), but one of the most attractive features of SNAP - leading SNAP to be used in the FAIN SNMP Activator - is it's high level of safety.

One security problem of most current systems is that packets may overuse local resources. The current approach to solve this problem is to allocate resource limits and watchdog timers, which implies that forced termination will be performed on any miss-behaving packets. Forced termination, however, is considered to be unsafe.

To avoid forced termination, SNAP uses type-checking and dynamic monitoring to against the packets damaging nodes and other packets - node should be able to predict the packet usage. Prediction is achieved by restricting the SNAP program to bytecodes and only forward branches are allowed. Under these restrictions each bytecode can only be executed once, at most. The effect of the restrictions is that both the CPU and memory usage of such SNAP program will be in a linear proportion to the packet's length, and the constant of proportionality is small and determinable. In other words, prediction of resource usage can then be performed based on the packet's length. Once prediction is made, the node can set its upper bound for the resource consumption of a particular packet.

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 SNAP will prove to be sufficient.

8.3.2.4 SNMP

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 version 3 provides cryptographically strong role-based access control.
- An extensible MIB and programmable SNMP version 3 agent have become available for conventional operating systems.
- Machines that run conventional operating systems are now sufficient 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 a operation codes for a programming language: LOAD and STORE. One could think of the object identifiers in the extensible MIB as memory locations. Simple programs can be written in SNAP to test the operational state and branch to different operation sequences.

8.3.2.5 Introduction to SNMP SNAP

The techniques of SNMP and SNAP are brought together in order to activate the SNMP daemon located on a node in a safe and efficient fashion. Essentially, the assemble codes in the raw SNAP packets are to be replaced with standard SNMP commands i.e. get / set. The SNAP packets embedded with SNMP command i.e. the SNMP SNAP packets are executed on the targeted active nodes on which both SNMPD and SNAPD are enabled. The embedded SNMP commands are then extracted from the SNMP SNAP packets and can then be used to perform standard network management on the active nodes i.e. altering the routing table.

8.3.3 SNMP Activator System Design

8.3.3.1 Injectors and Interceptors

8.3.3.1.1 Injectors

Injectors inject programs into the network to reconfigure it. An injector will decide to inject code, because it has intercepted a request for a data flow from its own network. An injector intercepts and interprets some part of an application protocol.

For example, the injector may intercept Network File System requests, obtain the user identification contained within the NFS request and use that to priorities the use of bandwidth to deliver the data. It can also make use of the MAC address, the IP address, 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.

8.3.3.1.1.1 Injection Strategies

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

Less often, 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. It would then be expedient to inject code into the remaining flows.

8.3.3.1.1.2 Experience with Injectors

An injector was demonstrated in the first audit: the ABLE platform used a packet filter to trigger the injection of code that constructed an IPSec tunnel. For the SNMP activator proposed here, a more sophisticated packet filter will be used. This will be a PromethOS system. PromethOS is preferred means of providing node operating system plug-ins for active nodes. It is extension of netfilter, a standard part of the Linux kernel. PromethOS packet filters will have a degree of feedback; they will be programmed by SNAP packets to wait for particular network events.

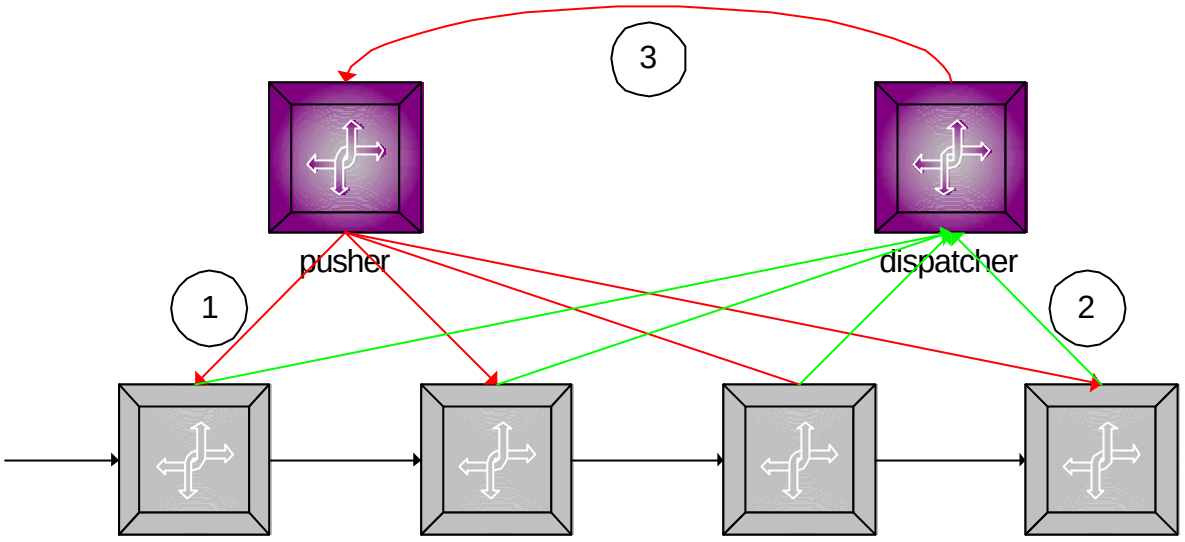


Figure 8-10: The pusher and the dispatcher

Code will be injected by the UDP SNAP packets. These 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. The 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. If this is the case, it would be desirable to implement another PromethOS module that performs packet spooling.

The difficulty with operating injectors is to decide what code to inject.

8.3.3.1.2 Interceptors

Intercepting SNAP packets is, technically, more complicated than injecting them. These are the constraints:

- 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.
- The functionality required will need to make use of active extensions on the node.
- Active extensions require blocked I/O.
- Blocked I/O cannot be performed in the same thread as the execution of the SNAP packet, because it would add too much latency.

Because of this a new invocation model is proposed.

8.3.3.1.2.1 Active Extensions

SNAP provides a facility to access services within the SNAP daemon: 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 SNAP daemon: 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 SNAP daemon and dispatched as the result in a subsequent SNAP packet.

8.3.3.1.3 Application Scenario: the pusher and dispatcher

The pusher and the dispatcher as shown in Figure 8-10 can model the injection and interception approach:

1. The pusher generates a special UDP packet (SNMP SNAP, embedded with SNMP command).

2. The pusher sends data to the routers to retrieve current network status.

3. The plain routers reply to the dispatcher with current network status.

4. The dispatcher receives the current network status from the plain routers, it then generates a SNMP SNAP packet to pusher. This SNMP SNAP packet then programs the routing table and sets the new default route.

8.3.3.2 Invocation Model

8.3.3.2.1 Kernel and User Space SNAP processors

The University of Pennsylvania is currently developing a kernel-based SNAP packet processor. This will be a node OS plug-in. This will be unable to invoke any active extensions outside of the kernel. Their proposed invocation model is this:

For two SNAP-enabled nodes: A and B.

1. A: Execute SNAP instructions that do not invoke active extensions.

2. A: On reaching an instruction that does invoke an active extension:

- A: Stop executing in the kernel and forward the packet to the next hop arriving at B.

- A: Continue executing the packet program in user space. Invoke the active extension, wait for the result and, when it arrives, send it onto the next hop as a SNAP packet that only contains the result.

3. B: the SNAP packet sent by A is now executed. Two conditions may arise:

- The result of the invocation of the active extension at the previous active node is required to progress the computation.

- It is not

4. B: If the latter is the case, the packet can continue to execute.

5. B: If the former is the case, then apply ii.

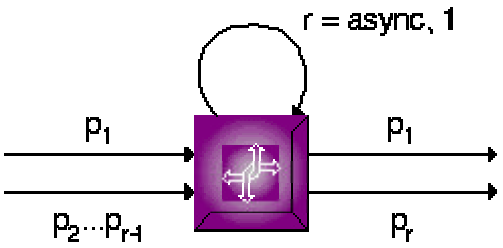


Figure 8-11: Invocation Model

In this way, SNAP packets can proceed very quickly through the network. A SNAP program will be in place at each active node waiting for the I/O to unblock at preceding nodes in the network. Diagrammatically, the situation is as given in Figure 8-11: Invocation Model. At time interval, 1, packet p arrives, denoted p_1. It blocking commands are invoked asynchronously and the packet is passed on. At time period p_r the result is ready.

Other packets arrive, p_2 though to $p_{r-1.}$. They may be forwarded or spooled. SNAP packets will almost certainly be dispatched, but it would be desirable to spool data packets . Eventually time period r arrives and the result of the SNAP operation invoked at time period 1 is available and it is dispatched immediately.

If the data packet flow is being spooled, a release indication would be sent by the next hop, presumably after it has received and processed the result of the operation of p_1 arriving after period r. A more sophisticated analysis than this would show that the synchronisation of the operation invocation and the arrival of the result forms a self-organising protocol - similar to Djikstra's leader election protocol for communications bus synchronisation.

8.3.3.2.2 Extensions to SNAP

In effect, the interaction between the kernel and user space SNAP interpreters requires two new primitives within SNAP: FORK and JOIN. These will be implicit in the calls to the active extensions: CALLS and SVCV. The design of the FORK and JOIN primitives is common to many operating systems. An identifier will be needed to specify the thread to join. The usual problem of finding a unique identifier in an open distributed system will be faced.

Also SNAP will require two stacks: a supervisor stack used for synchronisation and a user stack used for the SNAP program. The operation of the kernel SNAP interpreter will be an atomic copy, increment the program counter and forward, see the Booch object collaboration diagram given in Figure 8-12: SNAP-atomic copy, increment program counter and forward.

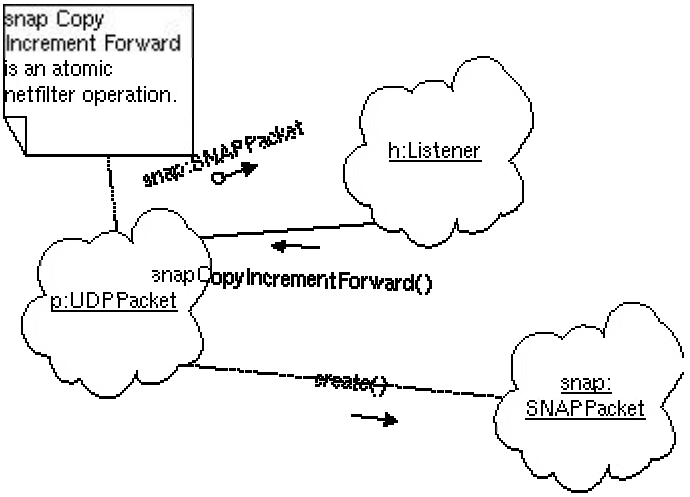


Figure 8-12: SNAP-atomic copy, increment program counter and forward

8.3.3.2.3 SNAP Active Routers Component Relationships

A Booch class diagram shows the expected relationship between the system components:

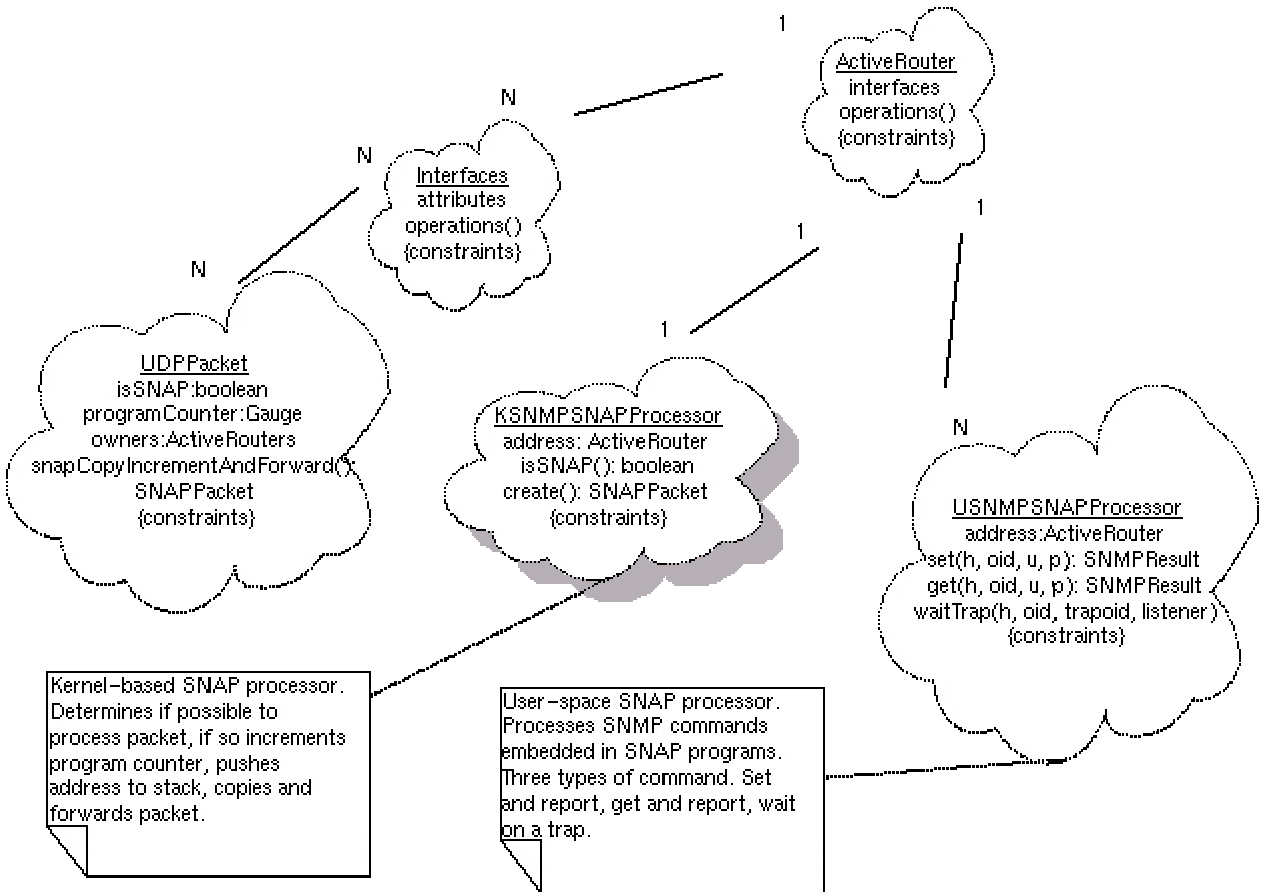


Figure 8-13: SNAP Active Router: Relationship Diagram

8.3.3.2.4 Proposed Implementation

The kernel space SNAP interpreter is not currently available, but the proposed invocation model (using a FORK and JOIN) can be proved using the current user space overlay network architecture of SNAP. Interceptors will be SNAP daemons running on active routers. They will listen on a number of 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.

8.3.4 SMNP-SNAP Application & Demonstration Scenario

The below collaboration diagram shows how a SNAP packet implementing a finite state machine could be used to create an ad-hoc network. There are four routers in this system: r, s, t, and u. Each of which must move to its respective operational state: s1, s2, s3, s4. The network to be constructed is a sub-network that passes all of its traffic through an ATM switch. The traffic must be conditioned so that the bitrate limited virtual channel carrying the traffic does not arbitrarily drop cells and corrupt the IP packets. To simplify the management of the traffic conditioning, the traffic is carried in an IP in IP tunnel and conditioned. It is then unencapsulated and given to the ATM switch's IP interface. Typically, this network might be used to support ADSL access for a neighbourhood.

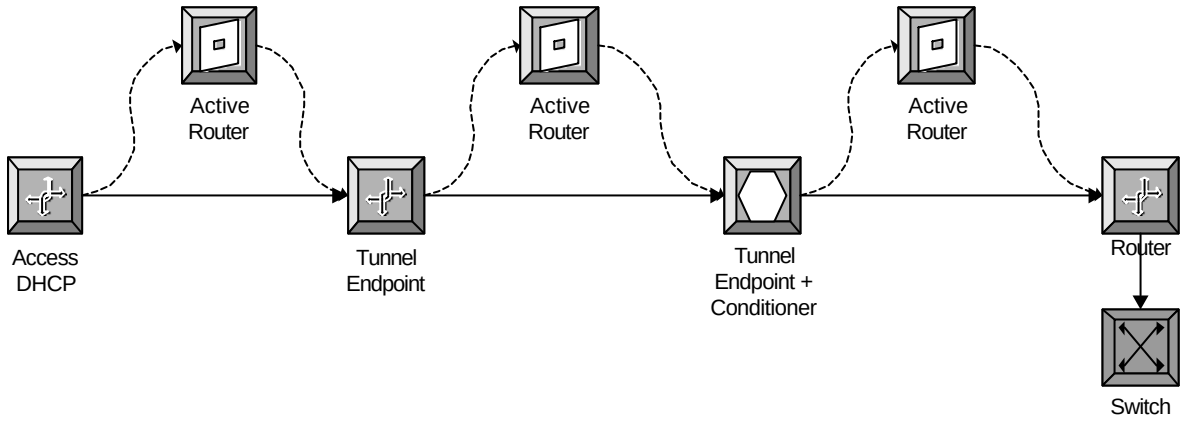


Figure 8-14: Network diagram for SNMP SNAP Application Scenario

The progression of the states of construction is this:

- S1 - The router r, supporting DHCP, RIP version 2 and, say, two 100BaseTx interfaces offers a sub-network to client machines. It creates an interface for that sub-network on one of its 100BaseTx interfaces and announces a route to the sub-network with RIP version 2 on the other interface. It injects the SNAP packet.

- S2 - An upstream router, s, receives the SNAP packet and is told to wait for a RIP version 2 event - the announcement of the new sub-network. In response it will create an IP in IP tunnel endpoint for it. It passes the SNAP packet on.

- S3 - The next upstream router, t, receives the SNAP packet and is told to construct the other IP in IP tunnel endpoint and to apply a traffic conditioner to the tunnel and to route the traffic to an ATM switch.

- S4 - The router with the ATM interface creates a route for the unencapsulated traffic of the sub-network.

This is the sort of network construction task that many system administrators must perform. SNAP is used to carry the instructions and to record the changes of state of the network. The instructions can be at conceptually a high level, the extensible SNMP agent allows many simple instructions to be grouped together. The states correspond exactly to the construction of the system.

Clearly, this task could be automated, the only variables are:

- IP sub-network to be supported
- IP tunnel endpoints addresses
- Traffic conditioning parameters
- ATM interface IP address

A set of each of these could easily be embedded into a number of SNAP programs. The DHCP routers would be given at least one each to inject into the network when a client machine starts to use the network they manage the addresses for.

8.3.5 SNMP Activator Conclusion

The proposed SNMP SNAP approach takes the advantages of both SNMP and SNAP: simple protocol, light-weight packets, safe, reliable, flexible and efficient. The combined approach enables source routing to be performed cost-effectively on plain routers as slow Linux source routing boxes are avoided.

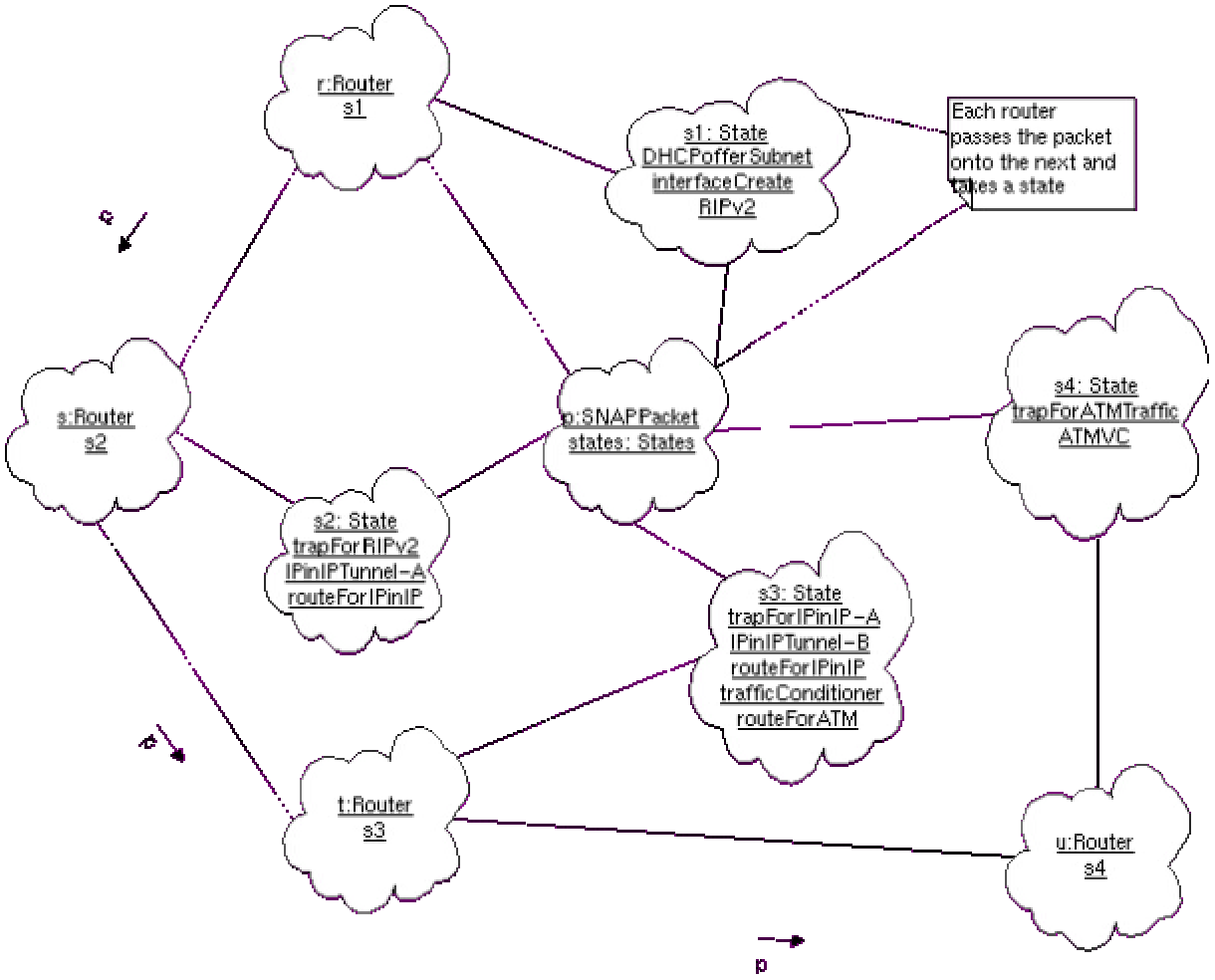


Figure 8-15: SNAP Program: ad-hoc network construction

The SNAP system described to propagate SNMP command execution through a network lends itself to mass production of SNAP programs to construct large numbers of network. It exploits active networking by having control information move with the data it must support.

The invocation model for SNAP presents the greatest challenge. If it is correctly developed to provide synchronised changes in state, disruption caused by transient operational states will be minimised.

The effect of the latter could be entirely eliminated with the use of packet spooling. This would be synchronised to an acknowledgement message that the network has attained its new state.

9 CONCLUSION & FUTURE WORK

This deliverable described the Revised FAIN Active Node Architecture the result of year 2 efforts. The original FAIN Active Router architecture has been refined and extended as a result of engineering decisions occurred during the implementation. Accordingly, security aspects of the system have been strengthened, and delivery of packets is currently done according to the virtual environment they belong to. The resource control has been integrated with the VE management framework and currently supports admission control decisions during VE instantiation phase.

We have created a framework that supports a component-based node architecture, which has raised the level of flexibility found in legacy systems. Along with this framework new concepts have been elaborated and old ones have been semantically enriched. These include the Execution Environment, Open Interfaces and the Network Element. We have also argued about how interoperability can be supported in a network environment that currently undergoes a lot of changes and the technology offerings are diversifying. This to our view is an important result that requires further investigation to assess its validity.

We have also built two different instances of the same type of EE that contribute towards component-based system architectures; one is Java based for quick prototyping and testing and the other is a high-performance EE supported by enhancements of a Linux-based kernel OS. The third instance of EE is a control based EE type that is deployed as an example of how signalling may be deployed to affect the communication behaviour as part of a service.

In conclusion, a prototype of a component-based AN node has been designed and implemented and its capabilities will be demonstrated during M3 – also described in R10.

During the third and final year of the project we will focus on extending the implementation work and further integrate between the different EEs. Some modifications are expected with respect to the architecture and the design of the FAIN Active Router with the most notable the RCF that should provide better isolation among the different Virtual Environments and enforce their corresponding resource profiles. In Appendix B, we provide a comprehensive table of the status of each one of the components together with their sub-components described in this deliverable. This table will impact the work that needs to be done during the next phase of the project.

There are several issues that have not yet been fully addressed in the design of the FAIN Active router and that should be covered in project year work and beyond. These are:

- Node interoperability plane: the control / management and transport planes separation controls the network element (which in turn controls the data-path creation), the control / management EEs can control the data-path behaviour. Different control / management EEs can implement the same open interface specification. The control and management interfaces could be exported inside different control / management EEs. A selected collection of all open interface specification of the control and management that defines the interoperability layer.
- Evaluation /porting of PromethOS and a selected set of node components facilities into a network processors environment for the creation of a high speed active router (type C node)
- Design & code optimisation of the node management framework for efficient management of the PromethOS and other facilities (i.e. efficient installation of code, efficient creation and management of virtual environments and resource control features). Provisioning of QoS enforcement facilities.
- Design & implement of a modular features set for the Resource Control Framework (i.e. enforce resource partition, admission control, resource control policy enforcement, export resource control interfaces)
- Design & implement a full features set for the Security Framework and integrated with the DEMUX and RCF (i.e. authorisation control, integrity checks)

- Design an optimised DEMUX facilities integrated with the Security and Node Management facilities including dynamic updating of Demultiplexing policies and transmitting packet data to an appropriate processing environment after classifying the data.
- Optimised the design of a higher-level mechanism for controlling the node management resources across a network. This mechanism is targeted to control uniformly the resources around the active nodes part of the active network.
- Dynamic updating of active services that involves reloading of classes from new code and restarting of instances.
- Evaluate the use of the FAIN Active Router components into a lighter version as Active Application Servers
- Evaluate Reference Configurations for efficient Active Nodes and Management deployment into a network.

Finally, the deployment of the AN node on a real testbed is also expected to have an impact on the work to be done as part of WP3.

10 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] H. Krawczyk, M. Bellare, and R. Canetti. Hmac: Keyed-hashing for message authentication. RFC2104, Informational, February 1997.

[3] FAIN D2 "Initial Active Network and Active Node Architecture" http://www.ist-fain.org/deliverables/del2/d2.pdf

[4] Sugih Jamin, Scott J. Shenker, Peter B. Danzig, "*Comparison of Measurement-based Admission Control Algorithms for Controlled-Load Service*" IEEE/ACM Transactions on Networking, 5(1):56–70, February 1997.

[5] BRESLAU, L., JAMIN, S., SHENKER, S., "Comments on the Performance of Measurement-Based Admission Control Algorithms" (IEEE Infocom 2000)

[6] Qin Zheng, Yasunoni Nemoto"Connection Admission Control for Hard Real-Time Communication in ATM Networks" (1996)

[7] McDysan, D., "QoS & Traffic Management in IP and ATM Networks", *McGraw-Hill Editions*, 2000

[8] Houatra, D., "Design of DENES, a Resource Control Support for Active IP Networks", *ICIN2001 – International Conference on Intelligence in Networks*, 1-4 October 2001, Bordeaux, France, pp.226-231.

[9] Houatra, D., "Distributed Control of Multicast Internet communications with Variable QoS Constraints", *SPIE ITCom2001 – Technologies, Protocols and Services for Next-Generation Internet*, 19-24 August 2001, Denver, Colorado, USA, *Vol.4527*, pp.102-112.

[10] Calvert, K. L. (editor), "Architectural Framework for Active Networks", version 1.0, 27 July 1999, available at: http://www.cc.gatech.edu/projects/canes/publications.html

[11] NET-SNMP community, http://net-snmp.sourceforge.net

[12] J. Moore "Safe and Efficient Active Packets", technical report MS-CIS-99-24, USA, Oct 1999. http://www.cis.upenn.edu/~jonm/

[13] L. Ruf, "Design of PromethOS", FAIN technical report, WP3-ETH-002-PromethOS, ETH Zurich, Dec 2001. http://www.promethos.org/

[14] J. Dittrich, C. Weckerle "ANEP Extension for Grasshopper", FAIN technical report, Germany, Oct 2001. http://www.grasshopper.de/

[15] Netfilter Core Team, http://www.netfilter.org

[16] R. Keller, S. Choi "An Active Router Architecture for Multicast Video Distribution", http://www.tik.ee.ethz.ch/~keller/infocom2000.pdf

[17] D. Raz, "An Active Network Approach for Efficient Network Management" IWAN99, July 99, http://www.cs.bell-labs.com/who/ABLE/

[18] M. Blaze, J. Ioannidis "Trust Management for IPSec", NDSS 2001, San Diego, Feb 2001 http://www.crypto.com/papers/

APPENDIX A

A.1 Active Packet Format for FAIN

A.1.1 ANEP Packet Format

	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 - - - n	Payload			

Figure A-1: FAIN Active Packet Format

Figure A-1 shows ANEP packet format. We have adopted an ANEP packet format as a FAIN active packet format. We only added options on 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.

A.1.2 Option Header Format

Figure A-2 shows an option format.

- **FLG**

It means how to deal the option data. If the value of bit 0 is one, that means the options are meaningful inside the specified execution environment. In addition, if the active node does not know how to process the indicated Option Type, the action taken is defined by the value of bit 1of the Flags field. If the value of bit 1 is zero, the option data is ignored and processing is continued. If the value of bit 1 is one, the packet is discarded. The owner of EE-ID defines the value of option type. In our case, we will set that the bit 0 is one and bit 1 is zero.

<table>
<tr><td></td><td colspan="4">0 31</td></tr>
<tr><td>N</td><td colspan="2">4N+ 0 Byte</td><td>4N+ 1Byte</td><td>4N+ 2 Byte</td><td>4N+ 3 Byte</td></tr>
<tr><td>0</td><td>FLG</td><td colspan="2">Option Type (14bit)</td><td colspan="2">Option Length (16bit)</td></tr>
<tr><td>1--m</td><td colspan="5">Option Payload (Option Value)</td></tr>
</table>

Figure A-2: FAIN Option Format

- **Option Type**

  It means a type of option. The owner of VE-ID defines the value of option type. Currently defined option types are shown in the Table A-1.

- **Option Length**

  It means length of option data. The length is shown in 32 bit words (4byte).

Table A-1: Defined Option Type

Option Type ID	Name of Option	Description of Option
0-100	Reserved	Those are reserved for future use.
101	VE ID	It is a field for identifying a VE ID.
102	EE ID	It is a field for identifying EE ID.
103-16383	Reserved	Those are reserved for future use.

A.1.3 Defined Option

A.1.3.1 Virtual Environment (VE) Identifier

<table>
<tr><td></td><td colspan="4">0 31</td></tr>
<tr><td>N</td><td colspan="2">4N+ 0 Byte</td><td>4N+ 1Byte</td><td>4N+ 2 Byte</td><td>4N+ 3 Byte</td></tr>
<tr><td>0</td><td>FLG</td><td colspan="2">Option Type</td><td colspan="2">Option Length</td></tr>
<tr><td>1</td><td colspan="5">Virtual Environment (VE) ID (32bit)</td></tr>
</table>

Figure A-3: Virtual Environment Identifier

Figure A-3 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.

Table A-2: Defined VE ID

VE ID	Description
0	Reserved
1	This number is assigned for privileged VE(ANSP) in a node.
Others	The number of VE-ID is assigned by ANSP(Network/Node owner).

A.1.3.2 Execution Environment (VE) Identifier

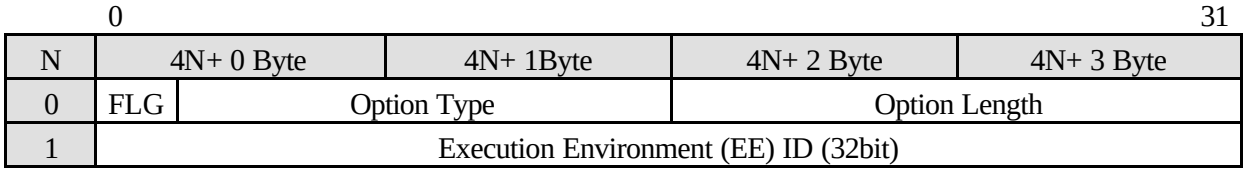


Figure A-4: Execution Environment Identifier

Figure A-4 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.

A.1.4 ANEP security related options and definitions

To enable operation of the security architecture, the architecture has to provide mechanisms to transfer security related information and provide data integrity service for active packets. Therefore we have chosen ANEP header as a carrier of needed information. ANEP header is the header of active packets. Options definitions, as will be presented in the following sections, follow the guidelines of the ANEP protocol.

A.1.4.1 Hop-by-hop integrity option

Hop-by-hop integrity option enables data integrity service for the active packet between two peer active nodes. It is defined as follows:

Hop-by-hop integrity is an option as defined in ANEP encapsulation protocol. ANEP option type for the option is 5. Option flags should be defined as public (1) in first flag and set for discard if not understood by active node (1) in second flag.

Option contains Key Identifier, Sequence number and HMAC. Key Identifier is 64 bit value and uniquely identifies sending or receiving peer on ANEP level. Sequence number is monotonically

increasing 64 bit number for a given Key Identifier. Sending system increases this value for every send packet. HMAC is a keyed hash either HMAC-MD5 (128 bits) or HMAC-SHA1 (160 bits).

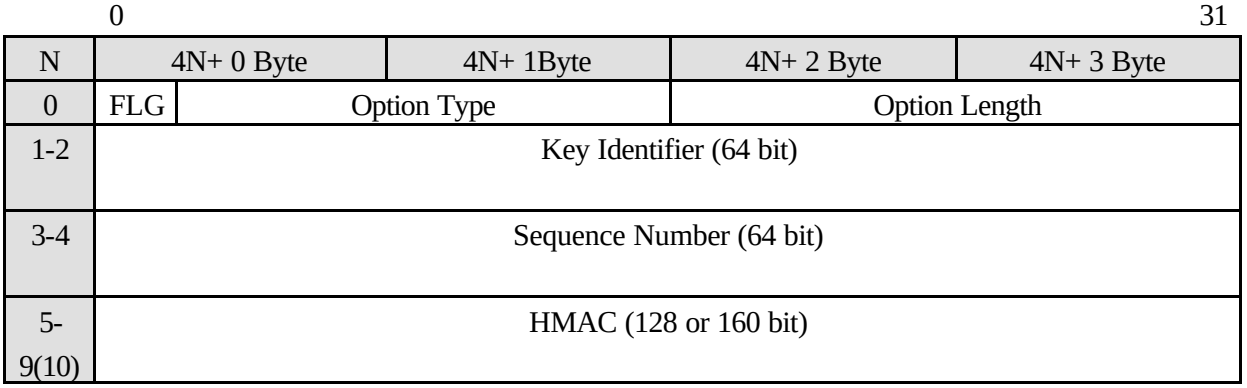


Figure A-5: Hop-by-Hop Integrity Option

HMAC in hop-by-hop integrity option covers whole ANEP packet except HMAC value itself. In the packet must be only one hop-by-hop integrity option.

While building this option the ANEP packet as will be send on the wire has to be presented in the security area. Depending on the information, where the packet will be send the right security association is selected. This way we can select the right key identifier, algorithm and secret key. Entire hop-by-hop option is build, with zeroed values for HMAC field. Packet is built as will be sending on the wire and the HMAC value is computed. Zeroed field in the option is replaced with computed HMAC value and the packet is returned to the MUX. Lower layer protocol fields are added to the packet and the packet is send to the next hop node. On the receiving side the ANEP packet has to be parsed so the hop-by-ho option fields can be extracted. On the basis of this information the right security association is selected with the needed dada to verify hop-by-hop integrity. For verifying the packet has to be rebuild again with zeroed HMAC field.

A.1.4.2 Credential options

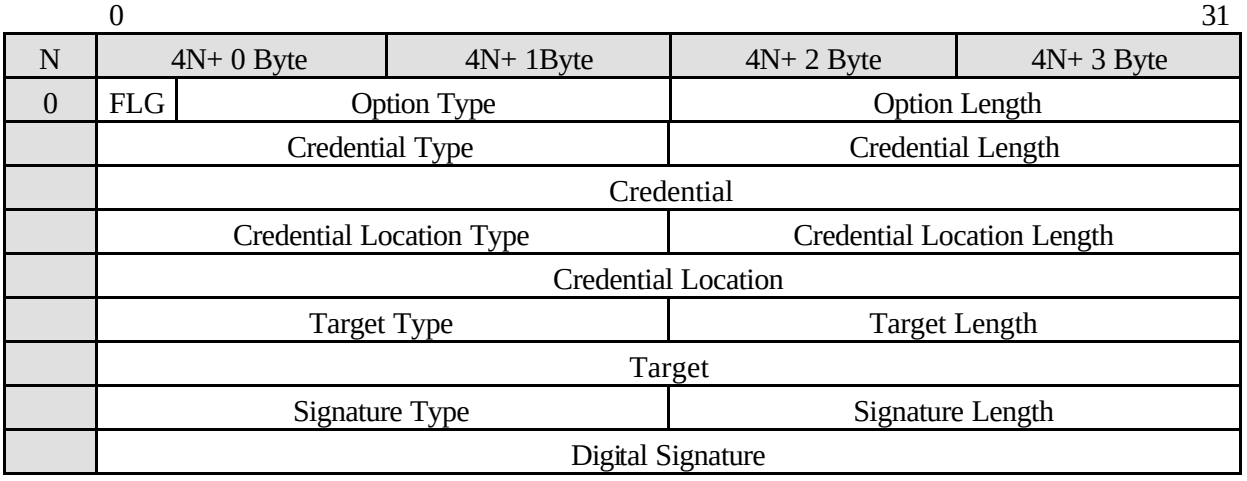


Figure A-6: Credential Option

Credential option enables data integrity service for the packet with a digital signature mechanism and binds credential option data to the packet. Credential option enables authentication service and can be used to transfer authorization information. Credential option is defined as is shown in Figure A-6:

Credential option is defined as an ANEP option. Its option type is 106. Option consists of credential, location and target triple in addition to digital signature. Credential payload carries credential or a reference to credential. Location payload points to the location where the credential can be get. Target payload specifies application domain of intended credential usage. Target length can be zero, so in this

case target payload is not defined. Digital signature covers credential option and static parts of the ANEP packet, at the moment only a payload.

There can be zero, one or more credential options in the packet. The triple and the digital signature are designed as sub options. For such sub options we have defined for now the following sub option types:

ANEP_SA_CREDENTIAL_TYPE_X509	1
ANEP_SA_CREDENTIAL_TYPE_X509_INLINE	2
ANEP_SA_CREDENTIAL_TYPE_X509_ATTR	3
ANEP_SA_CREDENTIAL_TYPE_X509_ATTR_INLINE	4
ANEP_SA_CREDENTIAL_TYPE_KEYNOTE	5
ANEP_SA_CREDENTIAL_TYPE_KEYNOTE_INLINE	6
ANEP_SA_CREDENTIAL_LOCATION_DNS	101
ANEP_SA_CREDENTIAL_LOCATION_WWW	102
ANEP_SA_CREDENTIAL_LOCATION_LDAP	103
ANEP_SA_CREDENTIAL_DS_TYPE_RSA	201
ANEP_SA_CREDENTIAL_DS_TYPE_DSA	202
ANEP_SA_CREDENTIAL_TARGET_NODEOS	301
ANEP_SA_CREDENTIAL_TARGET_EE	302
ANEP_SA_CREDENTIAL_TARGET_DOMAIN	303
ANEP_SA_CREDENTIAL_TARGET_AA	304

If there is no target option, credential in or referenced in the option should be used in all access control decisions.

The digital signature in the credentials option is covering the credential option itself, the packet payload and static ANEP options (VE and EE identifier). While the packet payload can change on the active nodes (being active code or reference to a code plus data plus headers) the payload itself has to be split into variable and static part. Therefore we need a place in the packet to store variable parts of the ANEP payload.

A.1.4.3 Variable option

Variable option is a portion of the active packet where the variable data related to active code or active application has to be stored. The option number for this option is 107.

Option flags should be defined as private (0) in first flag and set for discard if not understood by active node (1) in second flag.

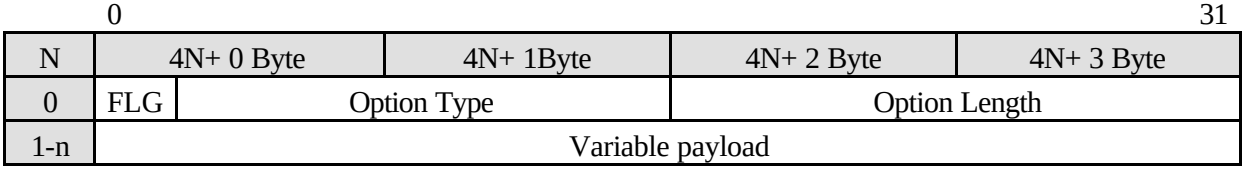


Figure A-6: Variable Option

A.1.4.4 Resource Vector

Resource vector option can be used to limit the scope of reach of the active packet in the network or as a hint for prediction of the resource usage on the node. For the same purposes simple TTL field in IP protocols is used. The TTL field is decremented at every IP node that the packet has passed. While the resource consumption in the active networks is more complex, the resource consumptions usage behaviour like proportional, differential an integral can be traced from node to node. Resource usage option as such is protected only by hop-by-hop integrity.

Format for the resource related option could be simple:

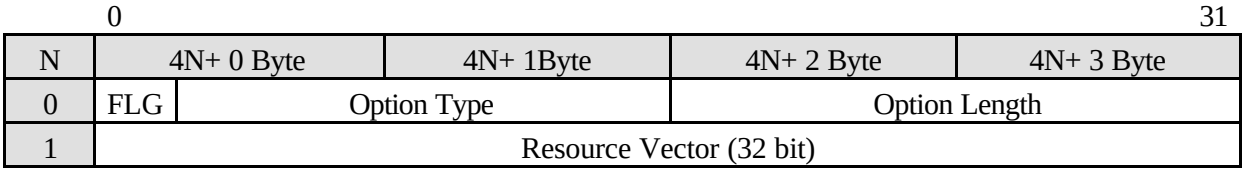


Figure A-7: Resource Vector

APPENDIX B

B.1 WP3 Components, Functionality and Status

Package / Component Area	Component Name	Lead Designer/Implementor	Participation in M3	Participation in M4	Participation in M5	Comments and size (cl.=#ofClasses, fct.=#ofFunctions, ncss=#of non-comment source statements, LOC=#ofLinesOfCode)
Active Node	Demux	HEL	Yes		Yes	Designed, developed, integrated, Size: 8 cl., 87 fct., 698 ncss
	Security (entry-level checks)	JSIS	Yes		Yes	Designed, developed, partially integrated, Size: 13 cl., 121 fct., 3649 ncss
	RCF	NTUA	Yes		No	Designed, partially developed, not integrated, Size: 5 cl., 27 fct., 406 ncss
	VE Management	FHG	Yes		Yes	Designed, developed, integrated, Size: 10 cl., 147 fct., 1643 ncss
	SNMP Activator	UCL	Yes		Noo	Designed, developed, partially integrated, Size: 2240 LOC
	PromethOS	ETH	Yes		Yes	Designed, developed, integrated, Size: 3101 LOC

Table B-1: Status of WP3 Components

B.2 WP3 Sub-Components, Functionality and Status

Working Area	Subcomponent Functionality	Implemented	Comments
DeMux	Channel Creation and Deletion	Yes	Creation of a channel instance and deletion of a channel instance.
	Filter Configuration Adding and Removing	Yes	Adding filter condition for receiving packet data and removing of the condition.
	Data Interception (Single Queuing)	Yes	Interception of data from a network based on a queue.
	Decoding	Yes	Decoding from byte stream to a packet object.
	Transmission	Yes	Data transmission from a network to a proper client.
	Encoding	Yes	Encoding from a packet object to byte stream.
	Retransmission	Yes	Data retransmission from a client to a network
	Security Function Call	Yes	Security function calls for receiving data and sending data to outside network.
	Channel Resource Reservation and Release	Partially	Resource is only managed based on full node resource. But its not managed based on resource that reserved by a VE.

	Multiple Accessing	No	This function is an enhancement of the DeMux. Data accessing by multiple clients.
	Data Interception (Multiple Queuing)	No	This function is an enhancement of the DeMux. Interception of data from a network based on multiple queues.
Security	ANEP packet, decoding	Yes	Available in Perl, encoding per specification porting to Java
	Connection Manager	Yes	Available in Perl, encoding per specification porting to Java
	Hop-by-Hop integrity sequences, sequence window	Yes	Available in Perl, encoding per specification porting to Java
	Authentication	Partially	Available in Perl, will be provided in Java
	Integrity by digital signature algorithm	Partially	Available in Perl, will be provided in Java
RCF	DENES	No	To be implemented partially next year
	Admission Controller. Node Level Admission Control	Yes	Performs admission check on node level by accessing the involving Resource Managers.
	Traffic Manager. Bandwidth Admission Control. Creation of the Traffic Classes.	Yes	The Resource Manager, which is responsible for the output Traffic. Performs admission check based on the availability of the output bandwidth and creates Traffic Classes for the VEs
	Traffic Class. Export Traffic Control Interface. Enforce the Bandwidth Partitioning.	Partially	Implemented: Export a control interface to the VE. Enforce the proper use of the part of the bandwidth that belongs to the VE. Enforce the Bandwidth partitioning per VE and per flow bases, by configuring the Linux TC. It gives a specific level of service to the flows. Next Year Planned: Calibration of the Linux TC in order to be able to assign specific amount of the bandwidth to the VEs.
	Interaction with Security	No	To be implemented next year
	Other Resource Managers	No	To be implemented next year
VE Mgmt.	JAVA EE	Yes	JAVA/CORBA environment used for executing VE Mgmt. and parts of Demux, Security, and RCF as well as active services. To take advantage of the implemented management features JAVA objects can serve as wrappers/proxies for non-JAVA implementations.

	Creation and management of VEs with associated resources	Yes	When a VE is created the VE manager tries to create all the required resources for this VE (EE, Security, Demux, etc.).
	Installation and management of active services	Yes	Uses a class loader to dynamically load code into a JAVA EE. VEs forward installation requests to the appropriate EE.
	Creation and management of service components	Yes	Dynamically instantiates objects in a JAVA EE.
	Dynamic configuration of service components	Yes	Service components can be configured via a flexible property scheme.
	Dynamic interconnection of service components	Yes	Service components can be interconnected via ports. Ports can exchange arbitrary information. There is implemented a strong support for CORBA ports.
	Monitoring of service components	Yes	Observers can register for notifications on changes of properties of service components.
	Authentication of clients of service components	Yes	Clients have to authenticate themselves when accessing a port. Clients can be distinguished by issuing client specific port references.
	Dynamic updating of active services	No	This should reload classes from new code and restart instances. This is subject of next year's implementation work. Planned by FHG.
SNMP Activator	SNAP - tuple to string conversion	Yes	
	SNAP - naming service lookup	Yes	
	SNMP version 2 "get" service	Yes	
	SNMP version 2 and 3 set and get services	No	Planned Project Year 3 by UCL
	Reset heap service	No	Planned Project Year 3 by UCL
	Kernel SNAP interpreter	No	Planned Project Year 3 by UPenn
	Kernel and user-space SNAP interpreters using hald-sync/hald-async negotiation	No	Planned Project Year 3 by UCL and UPenn
PromethOS	PromethOS Table	Yes	The PromethOS table is used to register at the different hooks provided in the Linux Network stack.
	PromethOS Management Component	Yes	The PromethOS Management Component acts as the management component of PromethOS in kernel space. It controls the plugins and dispatches packets to the appropriate plugins

	IPtables shared library	Yes	The iptables shared library is used to interface to PromethOS from user space to load the components and configure the filter expressions.
	WaveVideo Plugin	Yes	The WaveVideo plugin provides the active Video Scaling functionality.
	NAT Plugin	Yes	The NAT (Network Address Translation) plugin provides NAT functionality within PromethOS. It will be used to implement the Web Service Distribution case study as well as other services in the future.
	Resource Control	No	Planned for Project Year 3 by ETH.

Table B-2: Status of WP3 Sub-Components

