Masaryk University
Faculty of Informatics

DiProNN: Distributed Programmable Network Node

Ph.D. Thesis

RNDr. Tomáš Rebok

Supervisor: doc. RNDr. Luděk Matyska, CSc.

Brno, June 2009

Except where indicated otherwise, this thesis is my own original work.

Tomáš Rebok
Brno, June 2009

Acknowledgments

I would like to express my deep and sincere gratitude to my supervisor, prof. Luděk Matyska, for his guidance during my research—for his detailed and constructive comments, valuable advice, and important support throughout this work.

Further, I wish to express my warm and sincere thanks to Eva Hladká, head of the Laboratory of Advanced Networking Technologies (known as Sitola as well), who introduced me to the field of active/programmable networks, and who gave me an opportunity to become a member of both the Sitola laboratory and the "Multimedia transmissions and collaborative environment" research group of the Cesnet association. Moreover, I would like to thank her for the patience with me throught the tough times and her ability to keep me motivated during periods of self-doubt.

My deep and sincere thanks also go to Petr Holub for his help with the starting ideas in the beginning of my work, and for all of his assistance, which he has kindly provided to me in numerous ways during my research.

During my studies, I have worked with a great number of people, whose contribution in assorted ways to my research deserves special mention. I would like to thank Jiří Denemark, Lukáš Hejtmánek, Miloš Liška, Igor Peterlík, Dalibor Klusáček, David Antoš, and other fellows of the Sitola laboratory, who helped me a lot with various things during my research.

I would also like to convey my gratitude to all my friends for all their support and help I have received from them, as well as for all the enjoyable times we have spent. My special thanks go to Honza Petrželka, Tomáš Rybka, Zdeněk Vrbka, Michal Procházka, Mojmír Strakoš, Jaroslav Škára, and Terezka Obšilová. It is my pleasure to be your friend.

Furthermore, my deepest gratitude goes to my parents, grandparents, parents-in-law, and other members of my family for their unflagging love, patience, and support throughout my life: this thesis would be simply impossible without them. *"Mum, Daddy, there are no words I can thank you enough with…"*

Last but not least, I would like to give my special thanks to my wife Lenka, whose patient love enabled me to complete this work. In the recent years, she has been greatly helping and encouraging me in all the moments I was feeling down. *"Leničko, I am grateful to You for more than I could ever express…"*

And finally, my sincere thanks go to everybody that has been a part of my life, but whom I have failed to mention:

"Thank you very much!"

This work has been supported by the research intent *Integrated Approach to Education of PhD Students in the Area of Parallel and Distributed Systems*, No. GD102/05/H050, funded by Grant Agency of the Czech Republic, and by the research intent *Optical Network of National Research and Its New Applications*, MŠM 6383917201, funded by the Ministry of Education, Youth and Sports of the Czech Republic.

Tomáš Rebok

Abstract

The Active/Programmable networks allow the end users to inject customized programs into special network nodes, making them able to let their data being processed (in the way they want) directly in the network as it passes through. This approach has been presented as a reaction to a certain fossilization of the traditional computer networks, which on the one hand behave as a simple and extremely fast forwarding infrastructure, but which on the other have not been designed for fast and dynamic reconfigurations and novel services' deployment. Multimedia application processing (e.g., videoconferencing, video transcoding, video on demand, etc.), security services (data encryption over untrusted links, secure and reliable multicast, etc.), intrusion detection systems, and dynamically adapting intranet firewalls are just a few possible services, which could be provided.

Thanks to an amazing functional flexibility, the active/programmable networks became very popular in a short time and have been studied by many research teams. Various architectures have been proposed, from the integrated ones based on the active packets containing a program code (so-called *capsules*) to the discrete ones, where the program injection is separated from the processing of the data packets, all of them including software-only as well as software-hardware architectures. The fundamental issues, which have to be addressed by all the architectures, are:

- *Execution Environment Flexibility* – the active/programmable nodes have to provide an execution environment (EE), inside which all the user active programs (APs) are processed. Ideally, the nodes should be able to accept and run the user-supplied APs designed for an arbitrary EE, which will provide the highest flexibility possible. However, the existing solutions usually restrict the users to provide the APs designed just for a single and specific EE, ordinarily represented by a Unix/Linux-based OS, Java Runtime Environment, or a specialized proprietary one.

- *Resource Isolation and Security* – for security purposes, the running APs have to be strongly isolated from each other, so that a malicious/compromised AP cannot affect another APs sharing the same HW/SW resource(s) nor it can directly affect the simultaneously running APs themselves. Such an isolation has to further eliminate a hidden influence among the APs (e.g., through swapping of virtual memory pages) as well. Most of the architectures, which have been presented so far, more or less omit such security mechanisms at all, or provide proprietary mechanisms, which are externally enforcing defined security policies, but which do not address the fundamentals of the problem.

We claim, that instead of proposing novel and hopefully "more perfect" proprietary solutions, these issues could be generally addressed by making use of the virtualization techniques, which have revived in the recent years. And even further, besides helping to cope with these mentioned issues, the virtualization could also provide another useful benefits, which are discussed in this thesis as well.

The main goal of this thesis is to study and present the benefits of employing the virtualization principles in the active/programmable networks area. To illustrate them, we propose a novel programmable network node architecture, named *DiProNN (Distributed Programmable Network Node)*, that employs the virtualization techniques and makes use of their discussed features.

The employed virtualization, properly combined with the other desirable concepts, enables us to propose a flexible and powerful programmable node, which allows its users to develop their active programs for arbitrary execution environments and comfortably compose them into complex processing applications. Besides the execution environments' flexibility, the employed virtualization makes the proposed node further able to provide higher security and strong isolation capabilities, additionally enhanced by robust resource reservations and guarantees.

Contents

List of Figures

List of Tables

— To my parents and my wife Lenka —

Chapter 1

Introduction

Contemporary computer networks behave as a passive transport medium which delivers (or in case of the best-effort service tries to deliver) data from the sender to the receiver. The whole transmission is done without any modification of the passing user data by the internal network elements[1]. These "dumb and fast" networks became mature product where only speed is ever increased and there is no ambition except for simple forwarding of the data. This approach is more or less acceptable for common network services (emails, world wide web, etc.), however, it need not suit the requirements requested from the network by specialized applications/user groups. These could prefer changing such a simple but extremely fast forwarding paradigm into an active transport medium, which processes passing data based on data owners or data users' requests. Multimedia application processing (e.g., videoconferencing, video transcoding, video on demand, etc.), security services (data encryption over untrusted links, secure and reliable multicast, etc.), intrusion detection systems, and Intranet firewalls are just a few possible services which could be provided. The principle called *Active Networks* or *Programmable Networks* is an attempt how to build such intelligent and flexible network using current "dumb and fast" networks serving as a communication underlay.

A traditional computer network can be considered as a system whose end nodes provide computations up to the application level, while inner elements (routers, switches, etc.) provide computations up to the network level, and all nodes are connected via passive links. While the elements may be programmable to some extent, the control is always in the hands of network administrators. The major difference in the active networks is that the elements inside the network are directly programmable by the users, so that they can provide computations over passing data streams up to the application level. These inner elements are called *active nodes*, *active routers*, or *programmable routers* (all three with rather identical meaning). Users and applications have thus the possibility of running their own programs inside the network using these active nodes as processing elements.

Thanks to an amazing functional flexibility, the active/programmable networks became very popular and have been researched by many research teams. Various architectures have been proposed, from the integrated ones based on active packets containing a program code (so-called capsules) to the discrete ones, where the program injection is separated from the processing of the data packets, all of them including software-only as well as software-hardware architectures. However, programming of complex stream processing applications for programmable nodes is not effortless since they usually do not provide sufficient programming and execution environment flexibility. Usually, when a

[1] Even thought some network elements do interventions to passing data streams (e.g., firewalls, proxies, application gateways, etc.), these interventions are limited to processings defined by their administrator(s).

program performing requested processing exists, there might not exist a programmable node with an execution environment capable of running it (and vice versa), and thus the original program has to be revised or a new one has to be created.

To cope with such problems, the reinvigorated technology, that has become one of the most talked-about technologies in recent years—the *virtualization* [247]—can be helpful. By employing the virtual machines (VMs) principles, the flexibility of programmable nodes' execution environment can be greatly increased since the programmable routers become able to run completely different execution environments simultaneously—the ones based on common operating systems (e.g. Linux or FreeBSD) as well as the specialized ones uploaded by users. Moreover, the virtualization can also bring other benefits—e.g., strong isolation among the user VMs, security improvements, complex resource management, possibilities of enriched programming flexibility, etc.—and thus make the usage of the active/programmable nodes easier from the users' point of view, and safer from the applications' point of view.

From the users' point of view, another limiting factor of most active/programmable architectures, that have been presented so far, is the necessity to create and upload just a single active program (AP) performing required (and sometimes highly complex) functionality. The users are thus forced to create and manage complex applications—they usually do not have a simple way to compose the required complex functionality from several, single-purpose simple active programs, even though such a separation can bring both functional and non-functional benefits. For example, concerning the functional benefits, it can yield into better possibilities for the resource management system, since available resources can be apportioned for the set of active programs in a better way than for a single one (which requires the same amount of resources as the set of the APs in total). Thus, the particular single-purpose active programs performing the requested complex functionality could be spread over a distributed infrastructure, which can make the overall requested resources attainable even though these were not attainable for the original, complex AP. Regarding the non-functional benefits, it can make the node programming more comfortable, since it allows the applications to be composed from several independent components, which could be further reused, if necessary.

Last, but not least, the still increasing speed of network links and still increasing applications' demands for higher network bandwidths make the single-computer active/programmable nodes infeasible to process passing user data in real-time, since such processing may be fairly complex (e.g., video transcoding, data encryption, etc.). The node, which wants to be capable of processing higher number of active programs simultaneously running as well as to be capable of processing high volumes of data at high rates, has to distribute:

- *processing load*—to enable processing amounts of data that are impossible to process via any single computer,
- *network load*—to avoid bottlenecks formed by networking interface of a single processing computer.

If a computationally intensive processing is required, the distribution of the processing load is sufficient [126]. However, such a distribution is not suitable for many applications since a single computer's internal architecture may saturate when working with multi-gigabit data flows. Thus, in such cases, the network load needs to be distributed over multiple computers as well [127].

1.1 Contributions

The main goal of our work is to study and present benefits, which can bring the employment of the virtualization principles in the active/programmable networks area. Besides being discussed within this thesis, all the benefits are illustrated on a novel VM-aware programmable network node architecture, named *DiProNN (Distributed Programmable Network Node)*, which thus provides the following set of fundamental features (see the Section 3.2 for further discussion):

- **F1.** *The built-in execution environment flexibility* – the active programs, which can be processed by the node, could be provided for multiple arbitrary execution environments the node supports.
- **F2.** *Execution environments' uploading* – once the execution environments, which are provided by the node, are not suitable for the users because of any reason, they are able to upload their active programs encapsulated in their own execution environments, which the APs should be processed in.
- **F3.** *Component-based programming* – to simplify the node programming, the node is able to accept user sessions consisting of multiple single-purpose cooperating active programs (components) [255, 272] and data flows among them defined on the basis of the workflow principles [63]. The sessions' workflows could by dynamically adapted to changing conditions and, as results from the previous items, each component (AP) might be further designed for a different execution environment (provided by the node or uploaded by the user).
- **F4.** *Parallel/Distributed processing* – to make the node capable of processing higher amounts of data, its architecture is based on commodity PC clusters, allowing both parallel and distributed processing of the user sessions. The APs, which are intended to run in parallel, do not have to be adapted in any way to make such a processing possible.
- **F5.** *Complex resource management and QoS support* – to provide a different priority to different applications, users, or data flows, or to guarantee a certain level of a processing performance to a session, the node provides complex resource management capabilities, which allow the users to specify the resources required by the particular APs processing their sessions.
- **F6.** *Strong APs' and resources' isolation* – for security purposes, the running APs are strongly isolated from each other, so that a malicious/compromised AP cannot affect another APs sharing the same HW/SW resource(s) nor it can affect the simultaneously running APs themselves. Such a strong isolation also eliminates a hidden influence among the APs and ensures, that the APs cannot compromise each other in other way than through the network. Once the APs are strongly isolated, the accounting of the resources' utilization might be performed as well.
- **F7.** *Mechanisms for fast APs' communication* – since the processing components might want to communicate with each other (e.g., because of an internal synchronization and/or state sharing), and since such a communication should be provided as fast as possible, the node allows the definition of the control communication channels among the APs, which are provided by a specialized low-latency interconnect. The

APs themselves do not have to be aware of these interconnects during their development, so that they do not have to be adapted to a particular interconnect the different nodes' implementations use.

- **F8.** *Virtualization system-independent architecture* – the node architecture is neither designed for a particular virtualization system nor for a specific kind of them. It uses common networking techniques only, which allow the node to be built upon almost any existing virtualization solution.
- **F9.** *Hardware support for performance improvements* – to cope up with the overhead introduced by the use of virtualization, we also depict a FPGA-based programmable hardware network card's architecture, which allows accelerating the packets' manipulation inside the node.

The virtualization, properly combined with the other useful concepts, thus allows us to propose a very flexible and powerful programmable node, which allows its users to develop their active programs for arbitrary execution environments and dynamically compose them into complex processing applications. Besides the execution environments' flexibility, the employed virtualization makes the proposed node further able to provide higher security and strong isolation capabilities, additionally enhanced by robust resource reservations and guarantees.

1.2 Thesis Structure

The rest of this PhD Thesis is organized as follows: The Chapter 2 gives the state-of-the-art in the area of the active/programmable networks and presents several stand-alone and distributed programmable routers' architectures. It further introduces the virtualization techniques, and presents several existing virtualization solutions currently available. The chapter concludes by the presentation of several computer network's applications, which the virtualization principles have been successfully used for.

The Chapter 3 then depicts the benefits of employing the virtualization principles in the active/programmable networks area, and discusses the objectives, which the proposed node should satisfy (including the motivation for them). Finally, the chapter compares our work with existing (not-only) active/programmable nodes architectures.

The following chapter, the Chapter 4, provides details about the DiProNN's architecture, describing the functionality of all its units. It further discusses both the data and control interconnections the DiProNN uses, and the possibilities of its architecture modifications, that lead to its easier application by minimizing the amount of the units requested by the most general architecture.

The data transmission protocols, which are used for transporting the user data both inside and outside the DiProNN, are addressed together with the control protocols (the internal transmission ones used for communication of the user APs via the low-latency control interconnect and the external application-level one for the DiProNN management) in the Chapter 5.

The programming model, which is used for the DiProNN programming, is proposed in the Chapter 6, while the DiProNN's operational overview addressing all the processes occurring during the DiProNN's runtime (initialization, user requests, session establishments, data flows, etc.) is presented in the Chapter 7.

The following Chapter 8 then presents the possibilities of DiProNN's distributed processing, and provides an analysis of the scheduling problem (planning the APs onto the

available DiProNN nodes) requested by it. The chapter ends with the discussion of an application of the existing scheduling techniques for the DiProNN's scheduling.

The further DiProNN's features—the Quality of Service (QoS) support and the hardware support—are depicted in the Chapter 9, while the following chapter (the Chapter 10) shows several applications the DiProNN might be advantageously used for.

The Chapter 11 illustrates the DiProNN's independence on the virtualization system (by depicting its implementation in several existing virtualization solutions), while the following chapter—the Chapter 12—provides both qualitative and quantitative evaluation of the proposed architecture.

The semi-final Chapter 13 then presents an example usage scenario the DiProNN might be used for. The thesis ends with conclusions presented in the Chapter 14, the list of abbreviations, the list of bibliography, and the list of author's selected publications.

Chapter 2

State of the Art

2.1 Active/Programmable Networks

The programmability in network elements (switches, routers, etc.) was introduced in 90s of the last century as a reaction to a new requirement of enabling fast deployment and customization of future network services (for example, virtual networking [193]), for which the traditional computer networks have not been designed. The proposed concepts, named active and programmable networks, provide an architecture enabling these features by allowing network users to dynamically program inner network elements. The dynamic programming refers to allowing the users to inject an executable code into a network element in order to establish a new functionality at runtime. The basic idea is to allow third parties (end users, operators, and service providers) to deploy application-specific services (in the form of a code) into the network. [94]

As described in [51], two basic concepts emerged on how to make the networks programmable. The first one, supported by the Opensig community[1], which has been established through series of international workshops, argues that by modeling communication hardware using a set of open programmable network interfaces, an open access to switches and routers can be provided. This approach abstracts the network elements as distributed computing objects providing well-defined open-programmable interfaces (hence "programmable networks"), allowing third-parties to manipulate the network state using middleware toolkits (e.g., CORBA).

The other concept, supported by the DARPA[2] agency, constitutes several diverse active network projects advocating the dynamic deployment of new services at runtime mainly within the confines of existing IP [221] networks. The level of dynamic runtime support for new services goes far beyond the one proposed by the Opensig community—this concept considers the dispatch, execution, and forwarding of packets based on the idea of *active packets* (sometimes also called *capsules*). These contain a processing environment as well as data, that should be processed—at one extreme, a single active packet can prepare/boot a complete software environment seen only by that packet, while at the other one, each active packet can modify the behavior of available processing environment depending on its data needs. Such active networks thus allow a customization of network services on per-packet basis, offering maximum flexibility in a service creation, however, (compared to the Opensig concept) with the cost of adding more complexity to the programming model. Nevertheless, this approach is far more dynamic than Opensig's quasi-static programming of network interfaces.

[1] `http://opensig.comets.wisc.edu/`

[2] `http://www.darpa.mil`

However, both concepts share the common goal to go beyond existing approaches and technologies for construction, deployment and management of new network services in traditional networks. Since both concepts are forms of achieving an open programmability inside the networks, in recent years, the tendency has gone towards their integration. Thus, in this thesis we will assume both terms—active and programmable networks—to have rather identical meaning, representing the networks being able to accept and run (in particular execution environment) user-supplied active programs.

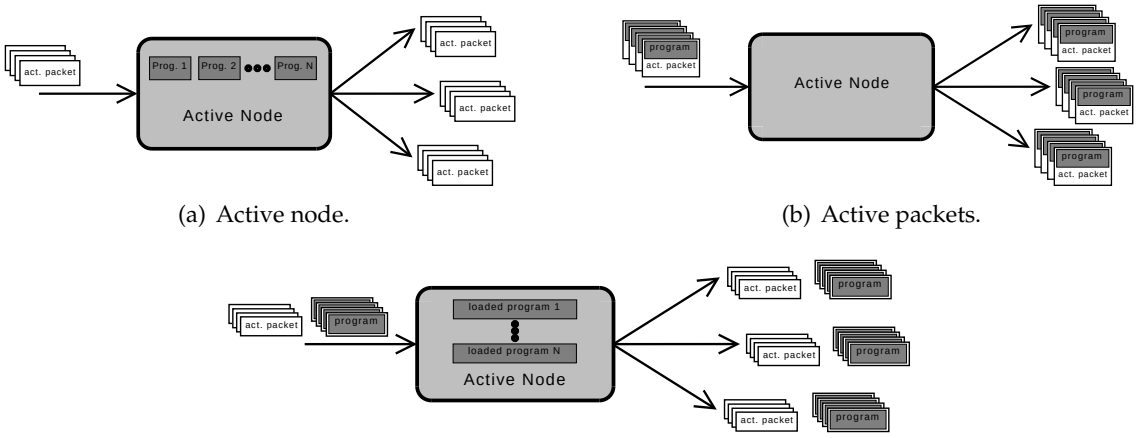
Active/Programmable Networks' Architectures
When considering the active/programmable networks' architectures, one possible classification criterion is the way an active code is delivered to them [222] (see the Figure 2.1):

- *Active nodes* – the active nodes approach (sometimes also called a discrete approach or a "plug-in approach") allows an injection of the AP's code into the active nodes separately from the data packets. The code can be injected before the data packets are sent (during an initial phase of the data transfer) or provided as node's built-in functionality. The main advantage of this architecture is that the code is injected only once, and thus its size is not limited and not critical. A disadvantage is the necessity to inject the code before the data transmission, which provides a larger startup latency, and a lower flexibility since it is usually hard to change the code during the data processing.

- *Active packets* – in so-called integrated approach, each data packet (the capsule) contains a program code, which is extracted on an active node and executed on the data part of that packet. This approach is very flexible since individual data packets—even the ones belonging to the same transmitted stream—can be processed in a different way. The node thus needs to be able to extract the relevant parts of the transmitted packets and execute the extracted code over relevant data. The disadvantage is that even a very limited extent of the code means a large overhead for transmitted data.

- *Active packets and active nodes* – the combination of both previous architectures allows the use of more complex programs while remaining flexible enough. Usually, a program is transferred before the actual data transmission occurs, but individual data packets contain some kind of parameters or specific program commands. This supports an individualized packet processing without the limitations of the active packets approach. However, the initial delay necessary to establish the session is not eliminated.

Node Operating System and Execution Environments
The active networking community has designed an architectural framework that defines a three-layer stack on each active node [48]. Based on the framework, the functionality of an active network node is divided into the *Node Operating System (NodeOS)*, the *Execution Environments (EEs)*, and end users' active programs/applications. The NodeOS represents the operating system's components implementing services such as packet scheduling, resource management, and packet classification, which are usually independent of a specific active networking implementation. These services are offered to the EEs running on top of the NodeOS—the EEs are thus isolated from the details of resource management and from the effects of behavior of other EEs.



(a) Active node.

(b) Active packets.

(c) Active packets and active node.

Figure 2.1: Active/Programmable Networks Architectures.

The EEs implement an active networking protocol-specific processing. They export programming interfaces which can be programmed or controlled by end-user active applications. Thus, an EE acts like a "shell" program in a general-purpose computing system, providing an interface through which end-to-end network services can be accessed.

Thanks to lots of possible applications (some of them have been mentioned in the introduction chapter), the active/programmable networks have become highly popular in a short time, and studied by many research teams. Various active/programmable nodes' architectures have been proposed—from the simplest ones providing a fixed functionality defined by nodes' administrators without any resource management and/or Quality of Service (QoS) guarantees, through the ones based on a specialized hardware to the complex ones based on a cluster computing and providing some kind of resource management and QoS guarantees. In the following sections, we describe several proposed architectures, focusing on the integrated ones as well as the discrete ones. Since some of them combine both principles in some way, we explicitly denote the ability to upload user's functionality (for integrated ones) or control processing on a per-packet basis (for discrete ones) in their description.

In the end of this section, we show several operating systems (NodeOSs) that have explicitly targeted programmable networks and finish with parallel/distributed active nodes' architectures related to our work.

2.1.1 Integrated Active Network Solutions

2.1.1.1 ANTS (Active Network Transport System)

The ANTS [276, 277], developed at MIT[3], is an approach based on a mobile code, a demand loading, and caching techniques. The ANTS architecture allows new protocols to be dynamically deployed at both routers and end-systems, without any needs for coordination and without unwanted interaction with co-existing protocols.

The ANTS node consists of an arbitrary router, the NodeOS (which is JDK over Solaris or Linux), the ANTS platform and, on top, the Java end-user application(s). The ANTS

[3] `http://www.mit.edu/`

platform comprises the node control component used for a registration of the ANTS protocols (i.e., the active services running on the ANTS platform), a component group responsible for handling both the active code and the state information associated with it, and a component group handling the data path for ANTS capsules (the ANTS routing).

The ANTS capsules carry parameter values for a related piece of a Java code. If the node, which a capsule is passing through, contains the related code, the node initializes the code with a capsule's parameter values and executes it. If the code is not present on the node, the node requests the code from the previous one (the node from which has the capsule arrived)[4]. This distribution/transport mechanism ensures that the ANTS nodes become primed "on the fly" by the capsules passing through them.

Concerning the resource management and the QoS guarantees, there is no support by the ANTS. There is no way to assign resources/priorities to the execution of capsules—the node just provides a simple computing network element, where the passing capsules could be processed in a best-effort manner.

2.1.1.2 PLAN (Packet Language for Active Networks) and PLANet

The PLAN programming language [117, 139], developed at the University of Pennsylvania, was designed and developed as a part of the SwitchWare project described later. The PLAN is a functional scripting language based on the simply-typed lambda calculus, which has been specifically designed for lightweight and simple programming of mobile code included in active packets—PLAN programs tend to be very small, so that they can easily fit inside the active packets. Furthermore, the ability to statically type-check the programs before active packets are injected into the network improves the safety of such a mobile code.

The PLAN considers a two-level programming architecture—the services for active packets are composed from the low-level functionality residing on the nodes using a high-level scripting language (PLAN). Thus, the PLAN provides a "glue" to create value-added and customized network services using low-level node's services (programmed in any general-purpose language). This distinction between the high-level (lightweight) and the low-level (heavyweight) programmability helps to design lean and efficient active programs.

Regarding a resource management system, the PLAN provides an explicit support for it, enabling the runtime environment to control resources' usage during the execution of active packets. Each EE counts the amount of processing and memory resources required to process particular packet—if the counter exceeds the upper bound for a resource before the AP terminates, it is immediately terminated by the system.

The PLAN programming language has been also used in the PLANet project [118] building an active inter-network, which implements network-layer services directly on top of the link layer[5] technologies. Although it is designed not to rely on the existing IP infrastructure, it is implemented as an overlay network based on the UDP/IP communication model for simplicity reasons.

[4]The Java programs loaded onto the ANTS node during the node start-up are called "code extensions". These cannot be uploaded directly by users and cannot be removed from the node during its lifetime. The distribution of such a code must be handled by some mechanism outside of the ANTS.

[5]The layers' specification can be found in the ISO/OSI Network model description in [288].

2.1.1.3 Smart Packets

The Smart Packets [240, 241] is a DARPA[6]-funded active networks project, which aims to apply active networks technology to network management and monitoring. The project argues that the active network technology is convenient solution for the network management, because it enables intelligent processing inside the network, closer to the network nodes/devices being managed. Compared to traditional systems relying on passive "polling" techniques to identify network problems, this approach improves communication efficiency as well as events' discovery.

To limit the complexity, the Smart Packets' nodes do not maintain a persistent state across the packets—they require the smart packets to be completely self-contained, without any connection states or fragmentations performed on the transport layer. Thus, the Smart Packets' programming language have to be able to fit active programs code under the MTU in length.

The project has further developed two programming languages—namely, the Sprocket and the Spanner. The Sprocket language is a high-level language much like C, but with security-threatening constructs (such as pointers) removed and special features necessary for various network management computations (i.e., special types for data packets and MIB access) added. The Sprocket programs are compiled into the Spanner—a CISC assembly language—which is further assembled into a compact machine-independent binary encoding, that is placed into *Program packets*—a Smart Packet dedicated to carry the code, which should be executed on the active routers along the transmission path. The other types of Smart Packets are the *Data packets* used to report the results of the program execution back to the originating network management program(s), the *Message packets* carrying informational messages rather than a code, and the *Error packets* used to indicate transport errors or execution exceptions. All the Smart Packets are encapsulated within the ANEP protocol [9].

When a Smart Packet arrives, a node checks it for integrity and performs authentication and authorization. If the arrived packet is a Program packet, an instance of an execution environment is instantiated and the code within the packet is executed. Otherwise, it is delivered to the appropriate user.

2.1.1.4 PAN

The PAN [204, 205], developed at MIT, is a kernel-based implementation of the active capsule approach. Although the mobile code is executed on a per-capsule basis, it is able to achieve high performance since it processes capsules directly in the OS kernel, performs minimal data copying, and caches the code for immediate execution.

PAN capsules contain both data they transport and a reference to a code object, that should be executed when passing a PAN node. The code object, that is provided either as a native Intel x86 code or a Java VM code, can direct the node to forward the capsule to the destination node, to modify the capsule's content, to pass the capsule to an application, or to access a state within the node. If the code object is unavailable on the particular node, it is dynamically loaded in a similar way as in the case of ANTS.

The PAN architecture supports multiple mobile code systems simultaneously running within a node (as a user-space processes within PAN's NodeOS). The prototype implementation supports a simple and completely insecure system for a dynamic loading of Intel x86 object code as well as an interface to a Java VM, which is, however, supported

[6] `http://www.darpa.mil/`

only in the user-space. The NodeOS is performed by any UNIX-like operating system—the node services then run either as a user-space process or a loadable kernel module. The PAN's resource management capabilities are also very minimal.

2.1.2 Discrete Active Network Solutions

2.1.2.1 Hladká, Salvet: Active Network Architecture

The Active Network Router [120, 122], developed at Masaryk University in Brno, uses the "active nodes" approach to active networking and the concept of "sessions" similar to connections in connection-oriented networks or sessions in the RSVP protocol [36]. The processing of a user code consists of two independent, but communicating processes. The first one controls the sessions' establishment and management, having the role of a control plane. It also incorporates a process of loading user functions into the router—the functions might be either pre-loaded (before or during setting up the connection) or loaded on demand during the data transmission (if a new requirement arises). The control process also provides book-keeping functions. The second process performs the packets' processing themselves—it executes the user code.

The generic design of the router's execution environment enables it to run different types of active programs designed for its Linux-based NodeOS—for example, compiled C or interpreted Java programs. The router management functionality as well as uploaded user programs are processed in the user space, whereas all the core of the router (schedulers, queue managers, etc.) are processed in the kernel space. The model also takes a resource management and network QoS guarantees into account, but these router's functions have never been designed in details yet.

The router has never been fully implemented, however, its main ideas were successfully used for a model and implementation of the user-empowered UDP packet reflectors [120, 121] to create a virtual multicasting environment as an overlay on top of current unicast networks. Moreover, the second-generation user empowered UDP packet reflector, called "Active Element", has been also introduced; to improve its scalability with respect to the bandwidth of each multimedia stream being processed, the concept of a distributed active element has been further proposed [126], which uses computer clusters with a low-latency internal interconnection to perform parallel/distributed processing.

The router's model has further served as a basis for protocol research and development, e.g., "Active Node Authentication Protocol (ANAP)" [72] and "Active Router Transport Protocol (ARTP)" [225], whose possible utilization in the proposed node is further discussed in the Section 5.1.

2.1.2.2 FAIN (Future Active IP Networks)

The FAIN project [94, 95] was a research and development project under the IST programme[7] (IST-1999-10561), partially funded by the Commission of the European Union. The project aimed to develop an open, flexible, programmable, manageable, and reliable (secure) network architecture based on the active node concepts.

The FAIN active node has been built upon a programmable network element, e.g., an IP router with open interfaces. The node's computing platform, consisting of a local operating system (NodeOS), one or more distributed processing environments (e.g., TINA-

[7] http://cordis.europa.eu/fetch?ACTION=D&CALLER=PROJ_IST&RCN=53059

DPE[8]), and system facilities (e.g., resource control framework) provides a layer through which downloaded/injected components communicate with the network and with each other. Upon the platform, a set of execution environments is created to host the service components.

The node platform provides the basic functions, on which the EEs rely. As such, the platform's lowest layer—NodeOS performed by the Linux, FreeBSD, or an embedded OS—manages node's resources and mediates demands for them, including transmissions, computing, and storage among the EEs. The EEs are thus isolated from details of resource management and from effects of another EEs' behavior.

A single FAIN node is expected to support multiple concurrent EEs[9]. Since different EEs may have different trust levels, the NodeOS protects them by enforcing boundaries and resource limits on each of its EEs. The EEs in turn hide most of the details of the node's platform from the users.

Further, to guarantee a secure and fair usage of resources, the FAIN platform includes a resource control framework partitioning allocated resources. The FAIN defines two types of resources—the *physical resources* referring to the node's hardware capabilities (e.g., CPU time, network bandwidth, memory, storage) and the *logical resources* referring to node's software capabilities (e.g., classifier table, computation table, filtering table, forwarding or routing table). All the resources are abstracted through the APIs used by the EEs.

The FAIN node's administrator is able to define a set of services (in fact, active programs), which can be used by the FAIN's users. In the FAIN network, the services are deployed via active service provisioning—the service providers are responsible for releasing new services as well as withdrawing the existing ones (in cases when a service update deployment or a complete removal is necessary).

2.1.2.3 ANN (Active Network Node)

The ANN project [69, 146], led by the University of Washington, is aimed at the design, prototype implementation, and demonstration of a high-performance active network node supporting network traffics at gigabit rates. The system is designed for maximum performance, which is achieved partly by running active programs in the kernel, but also by running on top of a dedicated hardware.

The top-level hardware architecture is based on the high-performance IP routing architecture [208], which was refined and optimized for the purpose of active networks. The node consists of a set of *Active Network Processing Elements (ANPEs)*, each one consisting of a general-purpose processor, a large FPGA, and a memory. The ANPEs are connected to an ATM switch fabric.

To utilize the ANN hardware architecture in an efficient way, the software architecture is optimized for high-performance as well. A highly efficient data path is provided—the high-bandwidth data pass through the components implemented in the system's kernel—whereas all management components are implemented in a user-space providing a flexible control path. The ANN supports two execution environments, namely the ANTS execution environment and the *Distributed Code Caching for Active Networks*, called DAN [70]. In the ANN, the NodeOS is performed by an optimized NetBSD operating system.

[8] `http://www.tinac.com/`

[9] As far as we know, just two different EE instances, namely the Java EE and the High Performance EE [147], have been implemented [71].

The code blocks implementing application-specific network functions are called "active plugins", which contain a code downloaded from the *Code Server*[10]. The code is downloaded as soon as a reference to it appears in a passing datagram; nevertheless, it can be downloaded by a special configuration packet or by the node's administrator as well.

The ANN node also contains a component responsible for the resource management functions, called *Resource Controller*. The component controls a fair CPU time sharing [284] and memory consumptions among active plugins on a per-instance basis.

2.1.2.4 NetScript

The NetScript project [245, 49], which was started in 1996 at Columbia University, was another pioneering project in the area of active networks. The project focused on a proposal of a programming model for the active networks, a programming language for network programming, and a programmable node architecture.

Regarding the programming model, the project has proposed a distributed programming model, where the script programs can customize node-resident functionality and services enabling them to define the processing of packet streams on individual network nodes.

The network architecture then compromises a collection of *Virtual Network Engines (VNEs)* interconnected by *Virtual Links (VLs)*. The VNEs—programmed by a script program (so-called *agent*)—process the packet streams and provide its services to other VNEs. The VNEs together with VLs determine a *NetScript Virtual Network (NVN)*. The NVN does not necessarily correspond to the underlying physical network—a single physical node might be responsible for executing several VNEs as well as a single physical link may relate to a collection of VLs (and vice versa).

The agents are defined using the NetScript programming language. It is a small and simple object-oriented dataflow language designed specifically for the programming of a stream-based computation [245]. To simplify agents' programming, the NetScript provides a library with a set of standard primitives (parse, flatten, split, join, etc.) and operations on streams of messages (multiplex, demultiplex, etc.). The agents can be deployed dynamically into the VNEs—a new code can be loaded and executed on-the-fly without disrupting the processing on the VNE.

The NetScript's distributed programming model enables the processing of a single stream by multiple agents—as soon as a packet arrives at a VNE, an information about the agents, which it should be processed with, is obtained from its header. This model combines benefits of both the discrete and the integrated approach to active networks—from one point of view, an active code can be uploaded onto VNEs as well as a node-resident functionality can be chosen by the passing active packet (discrete approach), while from the other, each active packet can define its own agent to be processed with (integrated approach).

2.1.2.5 Click

The Click [155, 156], originally developed at MIT with subsequent development at Mazu Networks[11], ICIR[12], and the University of California, is a modular software architecture

[10]The Code Server serves as a trusted, well known node for the plugins. It stores authenticated plugins only—each ANN can check particular plugin sources and its developer before it is installed on the node.

[11]`http://www.mazunetworks.com/`

[12]`http://www.icir.org/`

for building flexible and configurable routers. The Click router is composed from a set of processing modules (called *elements*) that provide packets with a processing functionality. In general, the elements (internally represented as C++ objects) examine or modify the passing packets in some way—each element conceptually performs just simple computations (e.g., IP packet's TTL field decrementing) instead of complex functionality. Click's configuration is determined at the compile time—the elements are inter-linked with each other using object references.

Each possible path for a packets' transfer is described as a *connection*—a link, through which the elements pass packets to one another. Click's configurations are then directed graphs of elements with connections as the edges. Each connection passes the data packets from an output port of one element to an input port of another one. The connections are of two types: *push* and *pull*. The push connection passes packets starting at the source element downstream to the destination element. In contrast, using the pull connection, the destination element initiates a packet transfer: it asks the source element to return a packet or a null pointer if no packet is available.

The Click is still being enriched by new features and supported by the author—the current Click's distribution[13] includes more than 300 elements, the Linux kernel module, the user-level driver, the FreeBSD kernel module, a driver for the NS[14] simulator, necessary service tools, and a documentation.

2.1.2.6 SwitchWare

The SwitchWare project [6, 10], developed at the University of Pennsylvania, uses three important components: the active packets, the active extensions called *switchlets*, and a secure active router infrastructure. The active packets are very similar to the capsules used in the ANTS project (described in the Section 2.1.1.1). The switchlets, which are written using the already described PLAN programming language, are dynamically loadable programs that provide node's specific services used by the active packets. Since PLAN programs are made secure by restricting their actions (e.g., the PLAN program cannot manipulate node-resident state), the active packets can call switchlets to compensate for these limitations.

The switchlets are modules written in the CAML (Categorical Abstract Machine Language) [171] language, which supports formal methodologies to prove security properties of the modules at the compile time. The code fragments, that are dynamically loadable and machine-independent, are authenticated by the developer and explicitly (not on demand) loaded into the switch.

At the lowest layer, the Secure Active Network Environment (SANE) [8] ensures the integrity and security of the entire environment. The SANE identifies a minimal set of system elements (e.g., a small area of the BIOS), upon which the system's integrity is dependent, and builds an integrity chain with cryptographic hashes on the image of the succeeding layer in the system before passing a control to that image. If an image is corrupted, it is automatically recovered from an authenticated copy over the network.

Even though the SANE has been developed as a part of the SwitchWare project, it addresses the problem of security for active network environments in general. It includes a secure bootstrapping mechanism providing static integrity guarantees [20], dynamic integrity checking mechanisms as well as a secure key and a certificate exchange mechanism enabling the code authentication [7].

[13] Available for download at: `http://read.cs.ucla.edu/click/`
[14] `http://isi.edu/nsnam/ns/`

2.1.3 Operating Systems for Active Networks

2.1.3.1 Genesis Kernel

The Genesis Kernel [159, 160] project, developed at Columbia University, has proposed a new class of open programmable architectures—so-called *Spawning Networks* [50]—supporting deployment and management of new network services. The spawning networks are capable of creating distinct virtual network architectures on demand—they consist of a parent virtual network capable of creating child virtual networks. The child virtual networks then operate on a subset of a parent network's topology and are restricted by capabilities of parent's underlying hardware. Further, it is isolated from the other spawned networks.

The Genesis Kernel is a programming system used for the creation, deployment, and management of new child virtual networks. It automates the virtual network life cycle and consists of several phases: the profiling phase, which provides a virtual network's design analysis, the spawning phase, which sets up the designed topology, allocates necessary resources, and binds transport, control, and management objects to the physical network infrastructure. The management phase manages virtual network's resources, and finally, the architecting phase on-demand adds, removes, or replaces distributed network algorithms.

According to [160], the Genesis supports spawning of virtual networks on three levels. The lowest one—a *transport environment*—delivers packets from source to destination end-systems through a set of open programmable virtual router nodes, called *routelets*. The routelets, which represent the virtual network, then constitute the lowest level of operating system's support dedicated to a virtual network—they process packets along a programmable data path at the inter-networking layer. The intermediate control level enables routelets interaction and controlling through distinct *programming environments*, while the top level is represented by the *binding interface base* [168]—an open programmable interface offering access to a set of routelets and virtual links constituting a virtual network.

2.1.3.2 JANOS (Java-oriented Active Network Operating System)

The Janos[15] project's [264] objective is to develop a local operating system for active network nodes oriented for executing an untrusted Java bytecode. The Janos primarily focuses on the resource management and control, the information security, performance, and technology transfer of broadly and separately useful software components. Conceptually, the Janos includes three major components of a Java-based active network operating system: the low-level NodeOS, a resource-aware Java Virtual Machine, and an active applications' execution environment.

Active applications for Janos are written using a slightly modified ANTS runtime (called ANTSR) running on top of a slightly modified, resource-conscious Java virtual machine, called *JanosVM*. These components constitute an execution environment's layer and run on the Moab [211]—an implementation of the NodeOS. The Moab enables precise specification of local node's HW resources to so-called *domains*—units of resource control similar to processes in traditional OSs.

Concerning the resource management, the Janos is able to limit the memory, the CPU usage, and the outgoing network bandwidth.

[15] `http://www.cs.utah.edu/flux/janos/`

2.1.3.3 Scout

The Scout operating system [196, 248], developed at the University of Arizona, is a configurable[16], communication-oriented OS targeted at network appliances (e.g., network-attached devices, set-top boxes, hand-held devices, and so on). An explicit *path*[17] abstraction [198] enables the Scout to effectively optimize the critical paths through layers of communication modules, resulting in a specializable operating system that has both a predictable and a scalable performance.

The Scout is configured by the *modules*, which provide a well-defined and independent functionality. Typical examples of modules are networking protocols, such as IP, UDP, TCP, or HTTP, or modules implementing storage system components, such as VFS, UFS, or SCSI. To form a complete system, individual modules are connected into a *module graph*: the nodes of the graph correspond to the modules included in the system, and the edges denote the dependencies between pairs of them. Such a configuration is defined at build time, and a number of configuration tools can assemble the selected modules into a Scout kernel.

The Scout further provides resource allocations and scheduling on a per-path basis—each path can define its requirements for an I/O bus bandwidth, for memory buffers required to absorb bursty data, and for data and instruction cache space to process the data without thrashing.

2.1.3.4 SPIN

The SPIN[18] [26, 27] is an extensible general-purpose operating system developed at the University of Washington. In contrast to the traditional approach, where applications live in user-level address spaces separated from kernel resources and services by protection boundaries, the SPIN enables applications to safely add system extensions into the kernel and specialize the running system depending on their needs. These kernel extensions (called *spindles—SPIN Dynamically Loaded Extensions*) can specialize the kernel (for example, add some services or replace default policies) or simply move an application into the kernel address space to achieve a higher level of performance. To secure sensitive kernel interfaces, the SPIN as well as spindles are written in the type-safe Modula-3 [54] programming language and are dynamically linked into the SPIN's kernel.

The SPIN kernel abstracts system's physical and logical resources, and implements a set of management policies for them. The low-level resource controllers provide light-weight abstractions of the physical hardware (such as page frames or activation contexts), while the higher-level resource abstractions (such as threads or address spaces) are implemented by collections of communicating spindles. The management of these resources is performed by the two-level resource allocation architecture—so-called *system allocator*, which manages a global pool of resources (such as pages, CPUs, or network bandwidth), and the *user allocator*, which manages private pools of resources that have been acquired from the system allocator. [26]

[16] In the sense that a given instance contains exactly the functionality required by the system for which it is built.

[17] In Scout, a path is an OS abstraction that encapsulates the data flow from an I/O source to an I/O sink. The path comprises two parts—a sequence of communication modules defining path's semantics (e.g., its reliable and/or secure transmission or other real-time behavior), and a collection of system resources required for processing and forwarding the data along the path (e.g., the CPU time, memory buffers, a cache space, etc.).

[18] `http://www-spin.cs.washington.edu/`

2.1.4 Distributed/Parallel Active Nodes Architectures

2.1.4.1 CLARA (CLuster-based Active Router Architecture)

The C&C Research Laboratories have proposed the JOURNEY network model [207] consisting of a network of routers with additional computational capabilities. The model defines that streams of active multimedia units (MUs) are injected into the network for routing to their destination as well as for customizing to the needs of their clients. Each MU consists of one or more packets and is computationally independent even of other MUs belonging to the same stream. It can be therefore processed independently on the rest of the stream.

A CLARA [274] is a prototype of a routing node in such a JOURNEY network. It consists of a cluster of generic PCs interconnected by a fast system area network (the prototype implementation uses Myrinet [32]). A single PC (the *routing element*) is configured as a normal IP router, while the others (the *computing elements*) provide computational resources for the customization services.

The routing element behaves as a normal IP router for incoming packets that have already been processed. Whether a packet is processed or unprocessed is determined by the fact, whether the IP Router Alert option [142] in the packet's header has been set or not. Therefore, processed packets are directly routed by normal IP router, whereas unprocessed packets are handed up to the CLARA software for possible processing[19]. The decision whether the packet will be processed or not may depend on CLARA's current conditions—whether the required processing functionality[20] and/or available computing resources are available.

The packets, that are accepted for processing, are dispatched to computing elements supporting required processing functionality. After the processing, packets are aggregated in the routing element and sent out through an appropriate interface (depending on their routing requirements).

The CLARA software framework is designed to support accounting of stream's resource utilization. The packets pass through multiple *stages*, where each stage comprises a sequence of modules, each of which encapsulates packet processing capabilities. Each stage accumulates a packet's resource utilization while it is being processed. The packet's cumulative resource utilization is then transferred along with the packet to its next stage. After this, any further resources, that are utilized by the packet in stages along its outgoing path, are accumulated at the stage associated with its stream.

Computational resources available in CLARA could also be reserved—the hierarchies of schedulers can be created allowing fine-grained divisions of computational resources available on the CLARA router. Thus, portions of router's computational capacity can be allocated to a user. The streams belonging to that user are then guaranteed that they receive at least the portion of the computational capacity that has been allocated.

Obviously, the JOURNEY network model does not process all the media units belonging to a particular stream at a single CLARA router. Moreover, the JOURNEY network does not guarantee that all the packets will be processed either—some of them may not be processed at all in cases when there are not enough resources to process them. Thus, additional guarantees must be implemented end-to-end, according to the requirements of individual streams.

[19] Since an MU may span several packets, the acceptance of the first packet of an MU necessitates the acceptance of all the packets belonging to the same MU.

[20] The CLARA router functionality is fixed (set up by CLARA administrator).

2.1.4.2 A Cluster-Based AR Architecture Supporting Video/Audio Stream Transcoding Service

The goal of this project [108] is to present a cluster-based active router implementation that provides video/audio transcoding service. As opposed to the JOURNEY network model described before, it is assumed that there is only a single active router on the media streams' path, where the transcoding is performed.

The proposed active router consists of a routing node and a few computing nodes. The nodes are organized as a distributed computing platform (a computing cluster) interconnected over a high-performance network (the prototype implementation uses the Gigabit Ethernet [68]). The routing node manages the whole cluster's functionality as well as behaves as a normal IP router.

Similarly to the JOURNEY/CLARA architecture, it is assumed that the media stream can be divided into a sequence of media units that are ready for independent processing (transcoding). The routing PC receives these media units and forwards them to the computing PCs for transcoding—each computing PC processes the MUs independently on each other using local computing resources. Because of the limited resources available, some MUs could be sent out of the router without being processed—the main aims of the router are to minimize the processing time of each MU as well as to preserve the MUs' order of outgoing stream as much as possible.

Similarly to another cluster-based systems providing the parallel processing, a critical issue of this router is an employment of a suitable load-balancing strategy distributing multiple media streams among different computing PCs to achieve the best utilization. As a part of this project, an evaluation of two load-balancing algorithms, namely, *round-robin* and *adaptive load-sharing*, has been done. To preserve the order of computations among media units as well as to keep the simplicity of round-robin, the project has proposed and implemented a new load balancing strategy—a *stream-based round-robin algorithm* [108], which, as the name indicates, sends all the media units belonging to a particular stream to the same computing PC.

2.1.4.3 LARA (Lancaster Active Router Architecture)

The LARA architecture [55] comprises both a hardware and a software active router's design. It consists of four parts, namely the *Cerberus* (a first prototype of the LARA concept), the *LARA Platform Abstraction Layer (LARA/PAL)*, the *LARA MANagement component (LARA/MAN)*, and the *LARA Run-Time execution environment (LARA/RT)*.

The basis of the LARA router architecture is provided by the Cerberus—a platform that can be built from common low-cost hardware components. The high-performance is achieved partly because of distributing computations over a cluster of high-performance processors, and partly because of the active processing performed in a kernel-space, which enables fast processing since expensive copy operations and context switches are avoided.

The LARA/PAL layer provides a platform independent layer able to provide various execution environments (for example, SwitchWare, ANTS, etc.) on various hardware platforms. It exports a set of programming interfaces (e.g., APIs to control and manage scheduling, memory, network bandwidth, and policy constraints) that can be used by the EEs.

The LARA/MAN then ensures the security on the LARA node. Depending on a policy infrastructure and an authentication mechanism, the LARA/MAN ensures that only

an authorized active code (the one uploaded from a trusted source or installed by an authorized user) will be loaded and executed by processing engines.

Finally, the LARA/RT execution environment is an extension of the Linux 2.2.x kernels. An active code designed for this EE must be written in the C programming language and provided either in the form of a loadable binary kernel module or a source code (later compiled)—the code loading/unloading is based on standard kernel module loading/unloading mechanisms provided by the Linux OS. To ensure a fair scheduling of the processing resources on a LARA node, LARA/RT includes a dedicated preemptive scheduler for active module threads. [233]

Although only a subset of the LARA architecture has been implemented, it has provided a useful input for the proposal of the LARA++ active router described in the following section.

2.1.4.4 LARA++ (Lancaster's 2^{nd}-generation Active Router Architecture)

As the name indicates, the LARA++ [233] has evolved from the LARA architecture developed at the Lancaster University as well. The reason to propose a new architecture has arisen from results of a usability study performed by the LARA team—they have realized that since the LARA requires "active programmers" to develop a low-level system code, it is not suitable for common users used to program in the user-space (i.e., different APIs must be used, the low-level system code is hard to debug, etc.). Further, the low-level system code has been highly critical with respect to system failures since every failure has typically led to a total system crash. [233]

As opposed to the LARA architecture, which has provided an innovative hardware architecture, the LARA++ focuses mainly on the node's software design that should be independent of the underlying hardware. This makes the LARA++ applicable on a single-processor as well as multi-processor systems, using both centralized or distributed router architectures.

The LARA++ components [234, 235] are dynamically loadable onto LARA++ routers, where they provide additional or extended services for individual data streams or even for whole protocol families. The components are built like normal shared or dynamically linked libraries and are distributed in the form of a pre-compiled machine code or a source code, which is compiled as soon as it is used for the first time (just-in-time compilation). To achieve security, the LARA++ executes active codes uploaded by users within restricted processing environments only, limiting access to low-level service routines and shared resources.

2.2 Virtualization Systems

The Virtualization and Virtual Machine technologies [247] have been developed in a number of contexts—programming languages and compilers, operating systems, and computer architectures—to enable new capabilities and to solve a variety of problems in interfacing major computer system components. In the area of programming languages, the virtual machines provide platform independence, supporting transparent dynamic translation and optimization; in the area of processor architectures, the virtualization principles mainly serve for an introduction of new instruction sets, while in the area of computer architectures, they usually provide dynamic optimization for power reductions, resource utilizations, and/or performance improvements.

Since the architecture of the proposed programmable node neither relies on a specific virtualization system nor on a specific VMs' type, we describe several VMs addressing the areas mentioned in the following sections. The description starts with the *platform-level virtualization systems* representing the most general VMs' architecture, which is followed by the *OS-level virtualization systems* and the *process-level virtualization systems*. An application of all the mentioned VMs' types in DiProNN is further discussed in the Chapter 11.

Because the virtualization technologies go through a purple period, many architectures have been proposed. Since their complete description is beyond the scope of this thesis, in the following sections we describe just the most famous ones. Finally, there are several examples of existing (non-programmable) network nodes employing virtualization principles given at the end of this chapter.

2.2.1 Platform-level Virtual Machines

The virtual machines for operating systems support (so-called *Platform-level Virtual Machines* or *System Virtual Machines*) were defined in the 1960s: in the IBM VM/370 environment, a virtual machine created an exclusive environment for each user [102]. The use of virtual machines is becoming interesting also in modern computing systems, mainly because of their advantages in terms of cost and portability [30]. Moreover, their ability to share resources in an effective way while maintaining a high degree of security is another factor, thanks to which the VMs are receiving a renewed interest after years of relatively little activity.

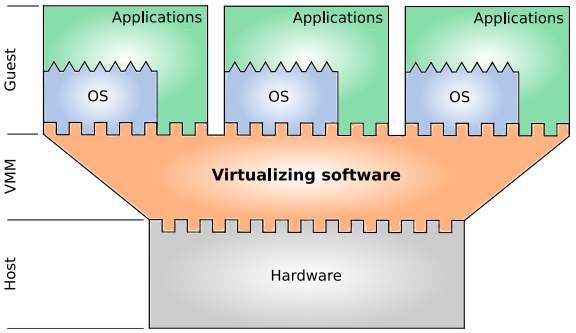
A virtual machine environment is created by a *Virtual Machine Monitor (VMM)*, which is sometimes also called a *hypervisor*, or "an operating system for operating systems" [145] (see the Figure 2.2(a)). The monitor creates one or more virtual machines on top of a single real machine—each VM then provides facilities for an application or a "guest system" that believes to be executing on a normal hardware environment. Thus, many processes, possibly belonging to multiple users, can coexist.

In the platform VMs, the major feature, which is provided by the VMM, is the platform replication—a platform (usually the one, which the VMM is running on) is virtualized and multiplicated, so that all the physical hardware resources are divided among multiple guest operating system environments. The VMM has an access to all the physical resources, which it manages—a guest OS and the application programs running inside it are then managed under a hidden control of the VMM; they are completely unaware of this "behind-the-scenes" work performed by the VMM.

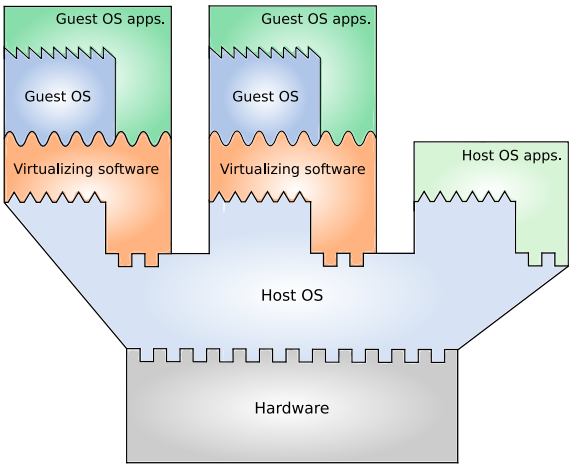
System and Whole-System Virtual Machines

The traditional system virtual machines provide the guest systems' environments with the same ISA (*Instruction Set Architecture*) as the underlying hardware, which the VMM is running on, has. However, there are situations, when there is a need to provide the guest systems with a different ISA than the host's one is. [247]

This situation has motivated system VMs, where a complete software environment (both an operating system and applications) is supported on a host system that runs a different ISA—such virtualization systems are called *Whole-System VMs*. Since the ISAs are different, the whole-system VMs have to emulate required guest ISA (e.g., via binary translation [247]), so that both the guest applications and the guest OS believe to run on their native computing systems. The most common implementation method of these virtualization systems is to place the VMM and the guest software on top of a conven-



(a) System virtual machines.



(b) Whole-system virtual machines.

Figure 2.2: Platform-level virtual machines [247].

tional host OS running on a particular hardware (providing native/host's ISA)—see the Figure 2.2(b).

2.2.1.1 QEMU

The QEMU [25], written by Fabrice Bellard, is a processor emulator providing the whole-system virtualization. It uses the dynamic translation[21] to achieve reasonable emulation speed—the converted binary code is stored in a translation cache and thus can be simply and fastly reused.

The QEMU uses two operating modes: a *full-system emulation*, which emulates a full-system architecture including several processors and various peripherals, and a *user-mode emulation* enabling QEMU to run particular processes compiled for a different CPU than the hosting one. In the full-system emulation mode, the QEMU supports many CPU architectures, including x86, x86-64, PowerPC, 32-bit Sparc, 32-bit and 64-bit MIPS processors, ARM and others.

[21] The *dynamic translation* is a runtime conversion of the guest CPU instructions into host CPU instructions. The advantage compared to an instruction interpreter is that the target instructions are fetched and decoded only once.

Since the QEMU source code is licensed under the GPL, it is often used as a basis for another virtualization systems of third parties—for example, Win4Lin Pro[22], VirtualBox[23], or KVM[24].

2.2.1.2 Xen

The Xen[25] [76, 112], initially created by the University of Cambridge Computer Laboratory and now developed and maintained by the Xen community as a free software, is a virtual machine monitor for IA-32 (x86, x86-64), IA-64 and PowerPC 970 architectures. It is licensed under the GNU General Public License [294].

The Xen provides a system-virtual machine environment, where the lowest and the most privileged layer is provided by the Xen hypervisor, above which one or more guest operating systems scheduled across physical CPUs run. The first guest operating system, called *Domain 0 (dom0)* in the Xen's terminology, serves as a Service Domain having special management privileges and direct access to the physical hardware. It is booted automatically when the hypervisor boots and serves for starting and managing of another guest operating systems.

The Xen provides mechanisms to manage resources, including CPU, memory and I/O. Among the other important Xen's features belong:

- **Paravirtualization** – the Paravirtualization is a specific type of the system virtualization, where virtual machines and the VMM co-operate to achieve very high performance for I/O, CPU, and memory virtualization. Thus, it enables high-performance virtualization even on architectures like x86 that are traditionally very hard to virtualize [312]. The paravirtualization exposes a virtual architecture that is a slightly different than the physical one. However, it requires guest operating systems to be ported (to use a special hypercall ABI[26] instead of certain architectural features) to be able to use it. Nevertheless, the user-space applications do not require any modifications.

- **Hardware-assisted virtualization** – in addition to paravirtualization, the Xen also allows unmodified guest OSs to be run on it. As opposite to traditional full virtualization hypervisors yielding in a tremendous performance overhead, the hardware-assisted virtualization, allowed by Intel VT (formerly Vanderpool) and AMD-V (formerly Pacifica) architecture extensions, can offer very high performance for para-virtualized guest operating systems as well as full support for unmodified guests running natively on the processor.

- **Virtual machine migration** – the Xen provides two types of migration used to transfer a domain between physical hosts—the cold/offline/regular one and the live one. The cold migration moves the virtual machine by pausing it on the source host and copying its memory content to a destination host, where it is resumed. The live migration [64] provides the same logical functionality but without needing to pause the domain—the domain continues its usual activities, so that the migration should be imperceptible from the user's point of view.

[22] `http://win4lin.net/`
[23] `http://www.virtualbox.org/`
[24] `http://kvm.qumranet.com/`
[25] `http://www.xen.org/`
[26] The *Application Binary Interface (ABI)* provides a program with access to the hardware resources and services available in a system.

2.2.1.3 VMware

The VMware[27] [186, 268] is a popular virtual machine infrastructure for IA-32-based PCs, initially based on the OS research at Stanford University. It is a commercial solution providing dynamic resource controls, complex resource management/reservation capabilities, high availability as well as backup tools making it the most robust and scalable virtualization technology currently available. It includes several products, e.g., the VMware ESX Server, VMware Server, VMware Virtual SMP, VMware Distributed Resource Scheduler, VMware High Availability, etc. [186]

The VMware products do not emulate ISAs for a different hardware that is not physically present. This provides better performance, however, can cause problems when moving virtual guests between hardware hosts running different ISAs as well as between hosts having a different number of CPUs. Nevertheless, the VMware technology is the most complex and sophisticated virtualization system on the market, providing most of the features available in the other virtualization systems altogether, including paravirtualization as well as offline and live migrations.

2.2.1.4 Denali

The goal of the Denali project [278, 279], developed at the University of Washington, is to propose a virtualization system capable of running large number of domains being simultaneously processed (tens of thousands domains on commodity hardware). The virtual architecture provided by Denali consists of three main elements—an instruction set, a memory architecture, and an I/O architecture.

The virtual instruction set has been designed for both performance and simplicity. The Denali's ISA consists of a subset of the x86 instruction set (enabling most of the virtual instructions to be executed directly on the underlying processor) enriched by a set of specialized instructions (e.g., the "idle-with-timeout" instruction helping to avoid wasting of the physical CPU by executing OS idle loops) and by set of virtual registers to expose system information, such as CPU speed, amount of memory, etc. However, the special virtual instruction set as well as the overall Denali's virtual architecture make it unable to run unmodified legacy guest operating systems. Even thought porting these OSs to Denali should be attainable, none of them has been really ported yet. Instead, the Denali's features have been evaluated using a novel operating system (called Ilwaco [278]), which has been proposed by the Denali project as well.

2.2.1.5 Other platform-level virtualization systems

Among the other platform-level virtualization systems, that should be mentioned, belongs the UML (User-Mode Linux)[28] [74]—a port of the Linux kernel being primarily developed by Jeff Dike since 1999. The UML provides a safe and secure way of running multiple Linux OSs within a normal Linux system. It does not provide machine emulation layer, rather, it provides virtual OSs (called *UML instances*) behaving as common processes in host's user space. The UML instances are not forced to run the same kernel version as the host runs—the kernel versions may differ, so that is entirely possible to test "bleeding edge" versions of the Linux OS on a system running a much older kernel. Moreover, the UML instances can be provided with a different hardware than the physical hardware is.

[27] `http://www.vmware.com/`
[28] `http://user-mode-linux.sourceforge.net/`

Furthermore, the KVM (Kernel-based Virtual Machines) [153] virtualization system, as compared to other systems, is a relatively new virtualization architecture utilizing additionally added virtualization extensions of Intel VT and AMD-V x86 processors, which enable the KVM (in fact, a modified Linux kernel) to behave as a relatively simple virtual machine monitor. Using the KVM, one can create and run several virtual machines behaving as standard Linux processes. However, unlike the UML, the KVM requires the physical host system to use the Intel VT or AMD-V x86 processors. The KVM also provides limited support to paravirtualization and live migration capabilities.

The last platform-level virtualization system, which is mentioned in this thesis, is the Bochs[29] [167]—a portable x86 and x86-64 IBM PC compatible emulator and debugger mostly written in C++ and distributed as a free software under the GNU Lesser General Public License. It supports emulation of the processor(s), memory, disks, display, network, BIOS, and another common hardware peripherals. It is able to run several guest operating systems (including the DOS, several versions of Microsoft Windows, BSDs, Linux, AmigaOS, Rhapsody and MorphOS) on many host OSs (like Windows, Windows Mobile, Linux and Mac OS X). However, since the Bochs provides a complete PC emulation, the guest system's performance tends to be very slow as compared to the other virtualization techniques.

2.2.2 OS-level Virtualization Systems

Especially because of the platform-level VMs' performance limitations, the lightweight virtualization systems—so-called *OS-level virtual machines*—have been proposed. They provide the OS kernel-level virtualization—a partitioning of the user-space environment of a single physical server into multiple small partitions (called *Virtual Environments*, *Virtual Private Servers*, etc.), so that each such partition looks and feels like a real server from the user point of view. Thus, instead of just a single user-space instance, multiple isolated user-space instances can be created, running on top of a single (core) OS's kernel (see the Figure 2.3).

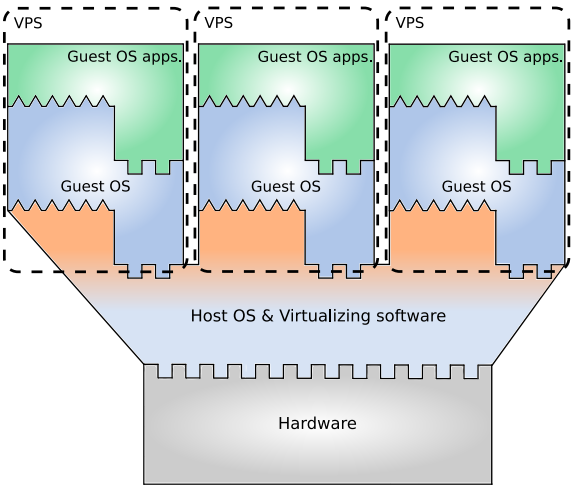


Figure 2.3: OS-level virtual machines.

[29] http://bochs.sourceforge.net/

The OS-level virtualization system provide a very low overhead that enables them to maximalize the efficiency of resources' usage—programs in a virtual OS instance usually use the operating system's normal system call interface and do not need to be subject to the emulation resulting in performance degradations. This negligible overhead enables the OS-level virtualization systems to run hundreds or thousands virtual environments on a single physical server—such level of density cannot be achieved by the platform-level VMs mainly due to an overhead of running multiple kernels. From the other side, the OS-level virtualization does not allow running different operating systems (i.e. different kernels) like the platform-level one, although different libraries, distributions etc. running on top of a defined OS kernel are possible. The single-kernel drawback can further represent a potential single point of failure as well as a security risk, if the kernel becomes compromised.

2.2.2.1 Linux-VServer and FreeVPS

The Linux-VServer technology[30] [176, 298] provides operating system-level virtualization capabilities to the Linux kernel (similar to OpenVZ described later). Similarly to the Xen, it is also developed and distributed as an open-source software and licensed under the terms of the GNU GPL [294]. As opposed to the Xen, the virtualization is not provided on the hardware layer, but on the kernel layer which requires all the guest operating systems to support defined kernel version.

The basic idea of the Linux-VServer technology is to separate the user-space environment into distinct units (called *Virtual Private Servers (VPS)*) in such a way that each VPS looks and feels like a real standalone server to the processes contained within it. The VPSs share the same system call interface and thus do not have any emulation overhead. This allows the VPSs to be able to run simultaneously on a single physical server at full speed, efficiently sharing hardware resources (CPU, I/O, memory, etc.) [298].

The requirement that the host operating system has to be able to run on a specific OS kernel belongs to main OS-level virtualization's disadvantages in general. Further, the VServer unfortunately does not provide any VPSs' migration capabilities[31] and, since the networking is based on isolation (not virtualization), it does not allow the VPSs to create their own internal routing and/or firewalling setups.

The *FreeVPS*[32], which is originally a fork of the Linux-VServer, is a GPL-licensed virtualization patch for the Linux kernel developed by Positive Software Corporation. Similarly to the Linux-VServer, the FreeVPS also provides complete isolation of the filesystem and processes running in each VM together with per-VM constraints for resources (e.g., network load, disk space, and memory consumption). In contrast to the VServer, the FreeVPS provides a variety of improvements in system accounting, resource management (limits on disk space, virtual/resident memory, the number of running processes, context file handles, TCP connections, etc.), networking, and other administrative enhancements.

[30] `http://linux-vserver.org/`

[31] In fact, there are some attempts trying to enrich the VServer with migration capabilities [75]. However, these were presented in a theoretical level only, since a real implementation rather remains still an issue.

[32] `http://www.freevps.com/`

2.2.2.2 OpenVZ

The OpenVZ[33] [254] is an open-source virtualization technology licensed under the GNU General Public License [294] and developed by SWsoft[34] as a part of their commercial virtualization system called *Virtuozzo*[35]. The OpenVZ creates multiple containers (also called *Virtual Private Servers—VPS* as well as in the Linux-VServer) running on top of a single kernel instance. From the kernel point of view, each VPS is a separate set of processes being completely isolated from each other.

The OpenVZ kernel is a modified Linux kernel adding the following functionality: virtualization and isolation of guest subsystems, resource management, and checkpointing. Regarding the resource management, the OpenVZ does a very good job in this area—over 20 crucial resources can be set live, while the VM is running, optionally saved to be re-initialized to the new value after a reboot [254]. The resources (such as CPU, memory, disk space, etc.) could be limited as well as guaranteed in some cases. Further, the checkpointing functionality allows the OpenVZ to provide both the *cold (offline)* migration as well as *live* migration capabilities [194].

2.2.2.3 FreeBSD Jails

The FreeBSD Jail mechanism [140, 189] is an implementation of the OS-level virtualization technology allowing the FreeBSD node administrator to partition the node into several independent mini-systems called *jails*.

The jails could be considered as an extension of the `chroot` command—a system call, which changes the root directory of a process and all its descendants. Processes created in the chrooted environment cannot access files or resources outside of it—once such a service is compromised, the attacker should not be able to compromise the entire system.

In contrast to it, the Jails are suitable for more complex tasks requiring a lot of flexibility and advanced features. They improve the concept of the traditional environment in several ways. In the chrooted environment, processes are only limited in the part of the file system they can access—the rest of the system resources (like the set of system users, the running processes, or the networking subsystem) are shared by the chrooted processes and the processes of the host system. The Jails expand this model by virtualizing not only access to the file system, but also the set of users, the networking subsystem of the FreeBSD kernel and a few other things. [292]

2.2.3 Process-level Virtualization Systems

Besides the platform-level and OS-level virtual machines providing a complete system environment for an (arbitrary) operating system, there are also the *Process Virtual Machines* capable of supporting just an individual process. These virtual machines provide user applications with a virtual ABI environment and usually serve as [247]:

- **Emulators and dynamic binary translators** – a challenging problem for the process-level VMs is to support program binaries compiled to a different instruction set than the one executed by the host's hardware is—i.e., to emulate one instruction set on a hardware designed for another. The most straightforward emulation method is an *interpretation*—an interpreter program executing target ISA fetches, decodes,

[33] `http://wiki.openvz.org/`
[34] `http://www.swsoft.co.uk/`
[35] `http://www.parallels.com/products/virtuozzo/`

and emulates the execution of individual source instructions. However, for better performance, the *binary translation* is typically used—blocks of source instructions are converted (translated) to target instructions that perform equivalent functions.

- **High-level language VMs for platform independence** – a cross-platform portability is a very important objective. A way of accomplishing this is to design a process-level VM environment at the same time as an application development environment is being defined. Such an environment does not usually correspond to any real platform—it is designed just for ease of portability and to match the features of the high-level language that it is used for.

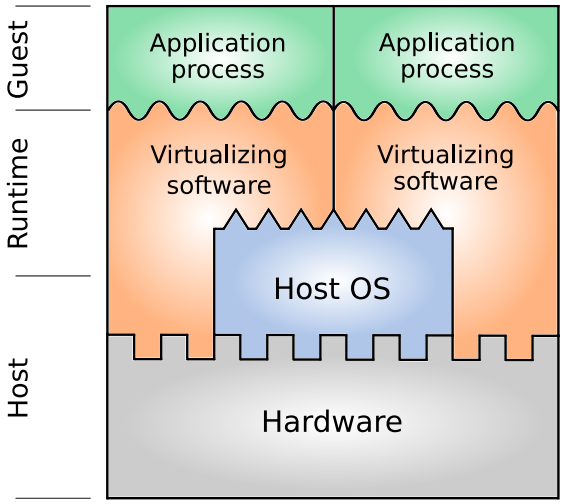


Figure 2.4: Process-level virtual machines [247].

One process-level virtualization system has been already mentioned in the previous section—the QEMU technology, which in the *user-mode emulation* mode behaves as an emulator of different guest ISA(s) than the host one is. The QEMU emulation is, as already mentioned in the QEMU description, realized using the dynamic binary translation. Among the other process-level virtualization systems belong:

2.2.3.1 JVM (Java Virtual Machine)

The Java Virtual Machine [178], developed by Sun Microsystems[36], is an abstract computer used for an execution of computer programs and scripts providing a virtual machine for high-level programming languages and serving as an instruction set simulator. The JVM operates on a Java bytecode, which is usually (but not necessarily) generated from a Java source code—the JVM can also process binary programs from other programming languages, if a proper compiler is used. For example, the Ada [23] or the Ruby [84] source code can be compiled into the Java bytecode and executed by the JVM. The availability of the JVM's implementations for various platforms makes the Java bytecodes portable across various platforms without any modifications or recompilations necessary.

The JVM provides so-called *just-in-time translation* [161] (also known as *dynamic translation*)—an alternative to the interpretation and the binary translation. Just-in-time trans-

[36] `http://www.sun.com/`

lator does not generate translated code (the code ready to be processed by the host machine) prior to program runtime (as the binary translators do[37]), but during it as the interpreters do. However, since the interpreters tend to be very slow, the JVM uses the just-in-time translation which translates an application into the machine code during its execution. The performance improvement over interpreters is reached by caching the results of translated blocks of code instead of a simple reevaluating each line/operand each time it appears. In contrast to the binary translation, the just-in-time translation can be further optimized to the targeted CPU and the operating system, which the application runs in.

2.2.3.2 CLR (Common Language Runtime) and Portable .NET

The CLR[38] is a core component of the *Microsoft's .NET framework*. It is Microsoft's implementation of the *Common Language Infrastructure (CLI)* standard, which defines an execution environment for a program code dedicated to the .NET framework. The CLR runs a form of a bytecode called the *Common Intermediate Language (CIL)*. Applications are written in any supported programming language, such as C# or VB.Net—at a compile time, a .NET compiler converts such code into a CIL code, whereas at runtime, the CLR's just-in-time compiler converts the CIL code into a code native to the operating system.

The goal of the DotGNU project[39], called *Portable .NET*, is to build a complete suite of free and open-source software tools (compilers, libraries, and tools) to compile and execute applications using the CLI standard. The initial target platform was Linux, however, the Portable .NET is working under Windows, FreeBSD, NetBSD, Solaris and MacOS X platforms as well. Further, since it is able to run on various processor architectures (for example, x86, PPC, ARM, Sparc, s390, Alpha, IA-64, and PARISC), it provides very extensive platform independence for programs written in C# and C programming languages.

2.2.3.3 LLVM (Low Level Virtual Machine)

According to [165], the LLVM[40] is a collection of libraries and tools that make it easy to build compilers, optimizers, just-in-time code generators, and many other compiler-related programs. It uses a language-independent instruction set both as an offline code representation (to communicate the code between compiler phases and to runtime systems) and as the compiler internal representation (to analyze and transform programs).

Among the strengths of the LLVM belong: extremely simple design, source language independence, effective optimization at compile time, extensibility, stability and reliability. Regarding the language independence, the LLVM includes frontends for, e.g., the C, C++, Stacker, and Java languages. Programs written in these languages can be compiled for and are able to run on the X86, X86-64, PowerPC 32/64, ARM, Thumb, IA-64, Alpha, SPARC, MIPS and CellSPU processor architectures.

[37] In fact, the binary translators do code generations both before and during the program runtime. They have to be able to translate a code during runtime especially in cases, when the code is available only at the runtime (e.g., self-modifying code).

[38] `http://msdn.microsoft.com/en-us/library/ddk909ch(vs.71).aspx`

[39] `http://www.gnu.org/software/dotgnu/`

[40] `http://llvm.org/`

2.2.3.4 Valgrind

Julian Seward's Valgrind[41] [242] is an example of employing process-level virtual machines for other reasons than the common ones based on platform independence or emulation. The Valgrind is a suite of simulation-based tools for memory debugging, memory leak detection, and profiling, designed for Linux OS and X86, AMD64, and PowerPC 32/64 processor architectures. The Valgrind's modular architecture consists of a core providing virtualized CPU in software and set of tools, each one performing some kind of debugging, profiling, or similar task. Among the standard tools included in the Valgrind distribution belongs: *Memcheck* tool detecting memory-management problems in programs, a cache profiler *Cachegrind* and a caller-callee profiler *Callgrind*, a heap profiler *Massif*, and the *Helgrind*, which detects synchronization errors in POSIX threading primitives.

In its essence, the Valgrind is a virtual machine, which provides a just-in-time binary translation—it translates the original program into a temporary simpler form called *Intermediate Representation (IR)*, which is a processor-neutral, SSA-based (*Static Single-Assignment*) form [173]. When the program is converted into the IR, the mentioned tools are enabled to do whatever transformations they like (for example, the Memcheck tool replaces the standard C memory allocator with its own implementation, which also includes memory guards around all allocated blocks—this enables Valgrind to detect memory reads/writes outside an allocated block). When the transformations are finished, the IR is translated back into the host machine code and run.

2.2.4 Virtualization in Current Computer Networks/Systems

Besides using virtual machines for platform's emulation, applications' portability, and/or platform multiplication for more effective usage of underlying hardware resources, the virtualization principles became very popular in modern computer networks, too. Especially because of their low cost and portability, the virtual machines have become employed in various applications, like intrusion detection systems or applications' development.

In this section, we show several types of networking applications, which make use of virtual machines benefits, starting with the section describing the idea behind several projects attempting to employ virtual machines in the Grid environment.

2.2.4.1 Grid Computing on Virtual Machines

The fundamental goal of Grid computing [149] is to multiplex distributed computational resources of providers among users across wide-area networks. Traditionally, these resources are multiplexed using mechanisms found in typical operating systems—for example, user accounts and time-sharings enable the multiplexing of processors, the virtual memory enables the multiplexing of the main memory, and the file systems multiplex disk storage. This approach, based on the operating system user level of abstraction, makes it difficult to implement security mechanisms necessary to protect the integrity of Grid resources as well as complicates the management of accounts and file systems not suited for wide-area environments. [82]

There are several projects [73, 82, 90, 143, 163] proposing to change the way, which the Grid computing is actually performing, by raising the level of abstraction from that of the operating system user to that of the operating system virtual machine. The VMs

[41] `http://valgrind.org/`

offer the ability to create a Grid environment on a host resource, which is optimized to suit particular users' needs—precise versions and flavors of operating system, libraries, middleware and applications can be deployed without any unexpected interference with other users' environments. The users thus get an illusion of having their own, dedicated virtual machine(s) customized to their needs.

Building virtual Grids over virtual machines provides also additional benefits—the VMs provide almost ideal encapsulation of the whole operating system and its components. This also allows the security improvements since a malicious user or an application can only compromise their own operating system within the VM, not the computational resource or another VM.

2.2.4.2 Cloud Computing

The Cloud computing [115, 273] is an emerging computing technology that uses the Internet and central remote servers to maintain data and applications. Data and processing power are stored in a shared *cloud* of Internet servers, and users access such an infrastructure when they want and in a way they need. The definition presented by [46] states, that:

"A Cloud is a type of parallel and distributed system consisting of a collection of interconnected and virtualized computers that are dynamically provisioned and presented as one or more unified computing resources based on service-level agreements established through negotiation between the service provider and consumers."

In fact, the Cloud infrastructure provides an illusion of infinite computing resources, which are available on demand. The key technology, that makes the Cloud computing possible, is the virtualization—the servers, which the infrastructure is built on, use the virtualization to apportion the applications and operating system resources to their clients. Moreover, it benefits from the virtualization's enhanced isolation capabilities as well—multiple guest operating systems can reside on the same physical hardware without any knowledge of the other ones, being thus protected from their instability and/or configuration issues. The dynamic relocations, instant rebalancing (moving VMs from over-utilized physical machines to lower utilized ones), and instant deployment are another virtualization features the Cloud computing benefits from.

Recently, several academic [21, 144, 180] and industrial [34, 137, 290, 295, 301, 303] projects have started investigating and developing technologies and infrastructure for the Cloud computing. The mostly used virtualization systems for building the virtualized infrastructure are the Xen and the VMware.

2.2.4.3 PlanetLab

The PlanetLab [62, 212, 213] is an open, geographically distributed platform for deploying, evaluating, and accessing planetary-scale network services. Its main goal is to serve as a testbed for developing large-scale distributed systems that can benefit from having multiple points-of-presence on the network. The project has been started as a reaction to the unavailability of any distributed system environment for distributed applications' developers.

The centerpiece of the PlanetLab architecture is a *slice*. Each deployed *service* (a set of distributed and cooperating programs providing some higher-level functionality) then runs in a slice—a network-wide container isolating services from each other. Each slice

encompasses some amount of processing time, memory, storage, and network resources across a set of individual PlanetLab nodes (machines capable of hosting one or more virtual machines) distributed over the PlanetLab network [62].

In fact, a slice is a set of virtual machines with each element of the set running on a unique node. Each virtual machine then provides an environment, where the program, that implements some aspect of a service, runs. Obviously, each VM runs on a single node and is allowed to consume some fraction of its resources. The PlanetLab implementation uses the Linux-VServer virtualization technique—a middle ground between the complete virtualization (e.g., Xen), which was considered to bring high price in CPU and memory resources, and the application-level virtualization (like JVM or CLR), which was considered to bring low flexibility.

The PlanetLab architecture has been used for several projects. For example, the *VIOLIN (Virtual Internetworking on OverLay INfrastructure)* project [135] has used it to create a prototype implementation of an architecture, which allows to create virtual isolated network environments (consisting of virtual routers, LANs, and end-nodes) on top of an overlay infrastructure.

2.2.4.4 XenoServer

Similarly to the PlanetLab, the XenoServer project [91, 113, 158], developed at the University of Cambridge, has also built a public infrastructure for a wide-area distributed computing. Its aim is to provide an infrastructure of computing nodes able to host virtual machines and scattered throughout the Internet network, allowing its users to submit and run their own programs to reduce the communication latency and avoid network bottlenecks as well as to deploy large-scale experimental services. Since the XenoServer allows VMs' migrations, it is also a suitable infrastructure for deploying mobile agent-based applications.

The proposed architecture consists of the XenoCorp servers and the XenoServers. The XenoServers, which run the Xen virtualization system, are hosted by remote machines around the Internet providing so-called *execution contexts*—execution environments, that run the services uploaded by the users. The XenoCorp servers then serve as a distributed directory for maintaining active XenoServers and registered users. Before starting the use of the infrastructure, clients have to register with XenoCorp. Once registered and properly authenticated, the clients can ask the XenoCorp to supply information about available XenoServers and their specification. Once a client selects the particular XenoServer as a suitable one for running the particular task, the deployment phase, which consists of two stages, follows—the session establishment stage, during which the resources required by the client are reserved, and the service deployment itself.

The clients define the execution context's requirements using a simple XML schema during the session establishment stage—they are able to choose an operating system, which their service will run in, from a set of supported traditional OSs published by the XenoCorp. The images of the OSs are prepared ahead of time and then deployed. It is expected that in most of the scenarios no user will ever need to login to the VM running the OS—the system boots up and starts required applications automatically. Moreover, to make the distribution of the execution contexts' images, that are required during VMs' migration, as well as the initial deployment stage easier, a novel wide-area filesystem, called *Xest* [197], has been created as a part of this project.

2.2.4.5 Virtual Networks

Many network services could benefit from having their own network topologies, direct control over the routing, forwarding, and/or addressing mechanisms rather than using a common communication infrastructure with other network services. For example, interactive applications (e.g., gaming, VoIP) can run application-specific routing protocols converging more quickly than existing network protocols, enterprises can construct and rent a private network connecting geographically dispersed sites, network service providers could run a separate "development" network for deploying and testing new configurations, protocols and/or designs, etc.

A software platform for hosting multiple virtual networks, called *Trellis* [28, 29], enables to define multiple virtual networks on a shared commodity hardware, allowing the users to define their own topology, control protocols, and forwarding tables. The Trellis makes use of two OS-level virtualization techniques, the Linux-VServer and the NetNS [299], together with the EGRE[42] tunneling mechanism, so that it provides a coherent platform enabling high-speed virtual networks.

A virtual network in Trellis is built using two components: the *virtual hosts* running user software and forwarding packets, and the *virtual links* transporting packets among virtual hosts. The virtual hosts provide an illusion of a dedicated physical host (even though multiple of the virtual hosts can run on a single physical hardware) allowing its users to implement both custom control-plane and data-plane functions without compromising the speed.

2.2.4.6 Virtual Machines for Intrusion Detection Systems

The *Intrusion Detection Systems (IDS)* are used to improve the security of computing systems. They continuously watch the system activity and look for attacks and intrusion evidences. In general, the IDS systems can be divided into two groups [11]: the *network-based IDSs*, which are based on watching the network traffic flowing through the monitoring system, and the *host-based IDSs* based on watching a local activity on a host (active processes, network connections, system calls, log files, etc.) [98]. The main weakness of the host-based IDSs is their relative frigidity—in order to be able to analyze the system activity data, the system has to run an agent collecting them. However, this agent can be deactivated or tampered by successful intruders in order to mask their presence.

The projects [98, 166] show, that this problem can be overcome by employing the virtual machines. The common idea behind them is, that the watched host runs in a virtual machine without any host-based IDS system installed. The IDS then runs in a different virtual machine (or directly in a service virtual machine) and monitors the activity of the host from outside. Thus, the IDS itself is kept safe since it is out of reach of intruders.

The prototype implementation of the [98] architecture, called *Livewire*, is based on the VMware virtualization system. Since their IDS executes directly on top of the hardware (on the hypervisor level), it scans only the low-level internal state of each VM being analyzed. Against it, the second approach presents interactions of the IDS VM(s) with the watched host VM, thus taking into account the activities carried out by its guest processes. The prototype implementation of this system uses the UML virtualization system.

[42] Ethernet over GRE [79].

2.2.4.7 Xen-Based Execution Environment

The main goal of the XenBEE project[43], which started in late 2007, is to create a *Xen-Based Execution Environment* [215] (hence the project name XenBEE) that allows its users to execute arbitrary batch applications (the applications without users' interactions) on a Grid-based remote computing infrastructure. The proposed architecture is based on the Xen virtualization system—the hosting node is able to accept user applications in self-contained virtual disk images (VMs), which have to contain the *XBE Instance Daemon* set to properly start appropriate application(s) inside the VM. The architecture thus allows to run arbitrary execution environments the host system supports—the particular EE has just to run the XBE Instance Daemon, which communicates with the main *XBE Daemon* controlling both the host node (including the communication with the XenBEE clients) and all the XBE Instance Daemons. The uploading (and running) the applications without any provided EE is not supported, since the XenBEE neither provides EEs for such applications nor it is able to upload an application into any of them—it focuses just on their proper startup.

The other XenBEE's execution semantic is the on-demand server deployment—a VM, which includes a server application, is supplied to the XenBEE and started in the same way as for the batch-based applications. However, as opposed to the batch-based applications, the server applications are supposed to run "forever" (to the time their VM is shutdowned) and to be reachable through a standard network connection (including VM's remote-login capabilities).

One of the XenBEE's goals is to be integrable with existing grid environments (e.g., the Globus toolkit [87], the Unicore [114], or the Condor [259]). Thus, the applications' specifications are described using a slightly modificated version of the *Job Submission Description Language (JSDL)* [17], which the mentioned grid middlewares support. The Public Key Infrastructure (PKI) is used for the authentication, authorization, and secure communication between the host system and the client. Nevertheless, the resource management and QoS capabilities have not been addressed by the project so far.

2.2.4.8 QuaSAR (Quality of Service Aware Router)

The main aim of the QuaSAR project [187, 188] is to demonstrate the feasibility of partitioning network router resources among independent flows by making use of virtual machines. The QuaSAR router consists of a number of virtual machines (called *(QoS) routelets*), each of which has an allocated portion of the underlying physical machine's resources (e.g., CPU time, network bandwidth, etc.). Each routelet is assigned to route a single data flow requiring defined QoS guarantees (e.g., defined using the RSVP signalling [36]) enabling the QuaSAR to provide QoS guarantees to each network flow individually. The VMM then ensures that each particular network flow can only access its allocated resources, and prevents the other network flows from interfering with it. One another virtual machine then routes the rest best-effort traffic and controls the router overall.

The QuaSAR prototype implementation has considered the use of Xen and Denali paravirtualization systems, however, because the Denali does not have any well-used operating systems ported to its virtual instruction set, the Xen has been used. The QuaSAR routelets run the Linux OS and route the MPLS traffic flows [230]. The routelets/VMs have necessary resources assigned by the Xen's virtual machine monitor—the CPU time

[43] `http://www.xenbee.net/`

is managed by the *Borrowed Virtual Time (BVT)* scheduler [77], while for the network transmission rates' allocations, the project has implemented a transmission limiting system (based on a simple credit allocation scheme) on its own.

2.2.4.9 SwissQM (Scalable WIrelesS Sensor Query Machine)

The Sensor networks [5] are a sensing, computing and communication infrastructure that allows to instrument, observe, and respond to phenomena in (usually) the natural environment. The sensors themselves can range from small passive microsensors (e.g, "smart dust") to a larger scale, controllable sensing platforms (used, e.g., to control the temperature and/or detect a toxicity in buildings).

The SwissQM [201, 200] is a sensor network platform based on virtual machines capable of executing bytecode programs on the sensor nodes. The SwissQM performs a combination of a software running on the sensor nodes and a software running on the gateway machine—the machine providing an access to the sensor network. The gateway machine translates user queries (expressed in various languages) into virtual queries (expressed in an internal format suitable for a multi-query optimization, query merging, etc.), which are later translated into network queries (expressed in a bytecode) and executed on the sensor nodes. This three-tier mechanism in combination with a flexible execution platform on the sensor nodes provides a level of abstraction hiding technical details from the users while allowing them to collect data in an easy, declarative way.

The *Query Machine (QM)*, which runs on the sensor nodes, is a virtual machine executing an instruction set—a small subset of the JVM extended with specialized instructions reducing the size of the programs.

Chapter 3

Motivation and Objectives

As already depicted during the introduction chapter, our work presented within this thesis focuses on investigating the VMs' features, which the active/programmable networks can profit from. At first, we present the intended benefits from more or less general point of view (the Section 3.1), while in the latter section, we study them together with another useful concepts in more detail (together with examples of real-life applications/situations, which can profit from them). The discussed benefits have thus become the objectives of our novel *Distributed Programmable Network Node (DiProNN)* architecture, that aims to verify their attainability.

Finally, the Section 3.3 briefly compares our work with the existing solutions described in the previous chapter.

3.1 Programmable Networks and Virtualization

For the active/programmable networks, the most straightforward virtualization's contribution is the possibility of running several platform-level virtual machines (and thus operating systems) simultaneously on a single physical network node. In an extreme scenario, one can think about a single physical node simultaneously running several active/programmable nodes, which have been described in the previous chapter, and thus behaving as a "multi-programmable" node. However, even being interesting, such a contribution does not provide any functional benefits except possibilities of more effective usage of the underlying physical hardware resources and thus is not important in the context of this thesis.

Execution Environments' Flexibility

The ability to run multiple virtual machines (with arbitrary OSs/EEs running inside) on a single physical node can, however, lead to the active/programmable node's ability to provide multiple distinct execution environments for user active programs. In this case, users are not forced to create their active programs for a single (and usually highly specific) execution environment, but they could be enabled to specify the EE, which their active programs require—the node can thus upload their active program(s) into a proper virtual machine running the requested environment. Moreover, all the virtual machines providing distinct EEs do not have to run all the time—the ones, that are not used at the particular time, could be made non-active (suspended) so that they do not utilize limited system resources, and re-activated (resumed) at the time an active program needs them. Even further, execution environments do not need to exist at all during the node

runtime—they can be assembled and run at the time an active program needs them, and later destroyed (stopped and deleted).

The active/programmable node, which is able to run multiple virtual machines simultaneously, can also enable its users to upload not only standalone active programs[1], but the whole virtual machine(s) with active program(s) running inside. Such a capability could be highly desired when:

- the node must not provide the particular execution environment (operating system) requested by the users' active program(s), e.g., because of licence constraints,
- users' active programs request highly specialized operating system's functionality and/or rely on an execution environment not commonly available (e.g., users' own execution environments),
- users' active programs require administrative privileges in order to run properly,
- the users do not trust the execution environments provided by the node for any reason,
- users' active programs have to cooperate with a shared service located in the same EE (e.g., multiple APs have to access (huge) data collections available in a database program, etc.).

Security and Strong Isolation Among User VMs/APs

Besides the benefits related to the execution environments' flexibility, the virtualization can also bring other features. For example, a strong isolation among the user VMs [181] and security improvements—what happens if an operating system or an active program inside a VM gets compromised and/or becomes malicious?

In common active/programmable nodes, a malicious AP could affect and compromise all the other APs running on the node as well as make the node unavailable for all the other users (by compromising the control plane). Such attacks are possible since the APs share the same execution environment in these common nodes—when an AP compromises the execution environment (an operating system) and acquires administrative rights[2], it is able to access the memory of all the other processes (APs) running on the node and compromise them.

In the case of two independent physical machines, when one of them becomes compromised, it almost always[3] has only one way to compromise the other—through the network. Such a security model is also available under the virtualization systems, since the execution environments of all the virtual machines do not share the same "physical" memory[4] and cannot compromise each other in other way than through the network.

[1] An active program uploaded without its virtual machine (execution environment).

[2] Compromising the execution environment and acquiring administrative rights is not always necessary—when an active/programmable node runs all the active programs under a single system user, the malicious process (AP) is able to access and compromise the memory of all the other processes belonging to the relevant user without any necessity to compromise the whole execution environment.

[3] Besides the attacks made through the network, the compromised machine can compromise the other, e.g., through a shared data storage (on an assumption that it somehow ensures, that the other machine executes a malicious program(s) uploaded there).

[4] In fact, all the virtual machines do share the same physical memory, since they run on the same physical machine. However, the VMM creates an illusion of an independent "physical" memory to all the running VMs (and their execution environments), that believe to run on an independent physical hardware with their own physical memory. Thus, if the VMM behaves correctly, the VMs and APs running inside them are not able to access the memory of all the other VMs, even if their execution environment becomes compromised.

Further, if we consider each virtual machine running just a single active program, we get all the active programs strongly isolated, so that they are not able to compromise each other. Even further, since the VMM can see and manage all the network communication among the VMs, in the case of each VM running just a single AP one gets absolutely isolated APs—when the VMM detects an unexpected communication among the VMs, it may suppress it as well as break all the unannounced communication by default.

However, ensuring that a compromised VM stays isolated requires a great deal of rigor and correctness in the VMM and all of the software in the host, that interacts with the VMs. This can be achieved through the use of sound security practices, so that one can reduce the risk of a compromisation of these components, and provide a greater assurance that the VMs stay isolated.

Complex Resource Management System

The strong VMs' isolation feature discussed in the previous paragraphs leads to possibilities of using a complex resource management system under the virtualized system. Once the VMs are precisely isolated from each other, they could be provided by a set of resources[5] (either requested or available at the moment), depending on the facilities of the particular virtualization system used.

Unlike typical multi-programming environments, where the resource control mechanisms are applied on a per-process basis, the VMs allow resource control at a coarser granularity—that of the collection of resources accessed by a user [82]. In traditional systems, the privileged control mechanisms interact with physical resources at the same level as the users' applications do, which requires the resource management system to be sophisticated enough, so that it is (among others) able to distinguish the privileged system calls, that imply from EE's actions, from that system calls, which imply from the user applications' actions. Against to it, the virtual machines are straightforward—once a user is provided with a whole "raw" machine, the resource owner sees a single entity, which he or she schedules onto his/her resources.

The provided resources could be further guaranteed—scheduled in a way satisfying running VMs' resource requirements—so that the VM is ensured that the requested resources will be always available through its whole runtime (and will not vary depending on the node's actual usage). In the case of each VM running just a single active program, such a node can provide resources for every particular active program running, providing a fairly complex resource management system (RMS). This RMS does not require any support of the APs' execution environments since all its functions are provided by the virtualization mechanisms, namely by the VMM.

Unification

So far, we have discussed the virtualization benefits from more or less straightforward perspective. Nevertheless, there is another aspect of mentioned benefits that should be pointed out—the unification of users' applications (the standalone active programs and/or the active programs running in the users' virtual machines) on a lower layer than in common active/programmable nodes (generally, in common non-virtualized computer systems). In common nodes, all the user active programs must be designed for an execution environment, which the particular node provides, since all of them have to run inside it. Thus, all the APs are unified in the sense that they have to satisfy EE's (in

[5] For example, CPU time, amount of free memory, network buffers and network bandwidth, storage subsystem access, etc.—see the Section 3.2.4.

fact, operating system's) facilities—for example, libraries, kernel versions, 32-bit or 64-bit architecture, etc.

However, in the case of System Virtual Machines (the Section 2.2), the user applications' unification is lowered to the hardware architecture layer—all the user applications have to satisfy the requirements of the hardware platform, which the particular VMM runs on, and which is provided for the VMs (especially its ISA). Thus, such a node does not require the users' applications to satisfy particular execution environment's requirements—the node's VMM just requires all the user applications to be able to run on a specific hardware platform. The VMM then serves as an execution environment for arbitrary execution environments (hence the VMM alias "an operating system for operating systems").

Even further, thanks to the Whole-system Virtual Machines, such a virtualized system does not require the user applications to satisfy its ISA at all, since it is able to emulate and run an arbitrary hardware platform. Anyway, in such a case, one also gets a layer all the user applications are unified on—the VMM layer.

Once the user applications are unified on a particular layer, the layer controlling process (the OS in common computer systems or the VMM in the virtualized computer systems) can manage and approach to all of them in a uniform way, which could be used especially for resource assignments. Moreover, the controlling process can manage the data flows as well as the communication among all the applications in a way it needs, while applications do not need to know about it. Such a data flows' management is also used in the novel programmable node we propose—see the following chapters.

3.2 DiProNN Objectives

The analysis depicted in the previous section identifies a set of features, which the virtualization can provide to the active/programmable networks. To verify their attainability, we propose a novel programmable network node architecture, which combines the virtualization's principles with another useful concepts allowing the node to provide a powerful and flexible programmable system.

In the following sections we discuss the objectives, that we require from such an architecture. Moreover, the real-life situations, which they could be profitably used in, are also depicted.

3.2.1 VM-aware Execution Environment Architecture

The beginning of this chapter depicts, that by employing the virtualization principles one can (among others) enhance the execution environments' flexibility of a computing system they are used in. Similar fact applies for the DiProNN—the virtualization can enable its users to upload not only standalone active programs later running inside some required execution environment (provided by the node), but they are also allowed to develop their programs for various specialized EEs, and to upload them together with the particular AP(s) encapsulated inside a whole virtual machine.

As the [45] states, such specialized execution environments can be customized to the needs of that particular application (active program) without any needs to support legacy interfaces, which the common general-purpose OSs have to provide. The overall performance of such a system can thus improve, because the applications run in execution environments, that are less complex and less demanding, and that are better suited to

their needs. The increased simplicity can also have positive implications on security and reliability issues.

For example, the high performance computing (HPC) applications were—based on the analysis depicted in [45]—observed to run significantly faster, when running stand-alone in a simulation environment than on top of a general-purpose OS in the same simulation environment. The tests published in [265, 266], which have been performed on a virtualized infrastructure, have shown, that the overhead of the virtualization results in slightly better run times for the native Linux OS at small workloads, but as the size of the workloads grows, a specialized stand-alone environment significantly outperforms such a general-purpose OS. Similar observations were also made in the XenSource project[6], which has presented a light-weight execution model for HPC applications [260] being able to run on top of the Xen hypervisor.

Another example, that can benefit from being able to be run in a specialized execution environment, are Java applications. Since the Java Virtual Machines provide their own implementations of scheduling, networking, and memory management, specialized execution environments could be used to avoid duplicating of these functionalities, which are also provided by the general-purpose OSs. This can allow the application code to run at a level much closer to the underlying hardware, which can result in an increased performance. The examples of such execution environments, which are specialized on running Java applications in a virtualized execution environments, are the Libra OS [12], the LiquidVM OS [300], the JavaGuest OS [138], the JX OS [104], and the JANOS OS (see the Section 2.1.3.2).

Furthermore, the applications requiring specialized light-weight and customizable operating systems, which focus on providing as high performance as possible—e.g., the Synthesis OS [185], the Synthetix [223], the Library OS [14], or the ones provided by the K42 project [162]—could be also supported by such a virtualized system. Last, but not least, the applications requiring a reliable and/or safe execution environments (e.g., the EROS OS [243] or the MINIX OS [257]), or the applications that do not trust the EEs provided by the particular node (e.g., the encryption/decryption applications depicted in the Section 10.5) could benefit from the ability to upload the whole execution environments as well.

As this brief survey indicates, the uploading of whole execution environments, which is not available in the active/programmable architectures presented so far, could be very useful for the users' applications, and thus the proposed programmable node should supports it. Basically, we have identified the following requirements (which cover the features *F1* and *F2*), that the DiProNN has to fulfil. The node has to:

- *enable its users to upload specialized execution environments for the APs* – as already depicted, the users should be able to develop their active programs for various execution environments having specific features (specific runtime software, libraries, kernel versions, supporting programs, etc.). Thus, the node should allow them to upload the whole execution environments encapsulated into the VMs.

- *enable its users to upload standalone active programs* – the node must not force users to upload the whole virtual machines only—the users must be able to upload an active program only (further referred as the *standalone active program*), which has to be further run inside an appropriate execution environment.

[6] `http://www.xensource.com/`

To allow this, the node should offer a set of built-in virtual machines, which provide various execution environments for the standalone APs (e.g., the Linux OS, the FreeBSD OS, the MS Windows OS, etc.). The suitable VM has to be chosen depending on the AP's execution environment requirements (e.g., based on the desired operating system or another system components the active program needs for its proper functionality).

- *provide built-in functions available for the users* – besides the uploaded APs/VMs, and besides the built-in VMs serving as execution environments for the standalone APs, the node has to be able to provide a built-in functionality (active programs) that might be used by its users. Such built-in functions should be set by the node(s) administrator(s) and should be available for multiple users simultaneously.

- *enable the administrators to run virtual machines with specialized functionality* – the node administrator should be able to run his own set of privileged virtual machines providing specialized functionality not related to a particular data flow (i.e., which may process all the users' data flows)—for example, anti-virus checks, classical routing and/or other network's/application's services.

- *enable running just a single active program per virtual machine as well as several active programs in a single virtual machine* – the approach of running just a single AP per VM, as outlined in the Section 3.1, brings several benefits (for example, the ability to employ complex management system enabling resource allocations for individual APs). The node has to support it, however, it has to be able to run several APs per VM as well (especially because of efficiency purposes).

The drawback of employing the virtualization is that it brings some performance overhead necessary for the VMs' management [192]. This overhead depends on system characteristics (including the processors' ISA, the VMM architecture and implementation, etc.), and is especially apparent for I/O virtualization, where the VMM or a privileged host OS has to intervene every I/O operation. We are aware of these problems, but we have decided not to restrict the DiProNN's design to the actual state in the virtualization systems—we believe, that lots of current issues will be solved in close future. Nevertheless, we have also decided to examine, whether there is a possibility of employing an accelerating hardware in the proposed DiProNN node, which can lower the virtualization's performance overhead as low as possible (the feature *F9*).

Last, but not least, the node architecture should be general enough so that one should be able to choose a virtualization system, which suits best the requirements requested by a particular implementation (the feature *F8*). The other reason for building such a general architecture is, that if the node had been designed for and/or dependent upon functionalities of a particular virtualization system, which later becomes unsupported or which becomes unimportant because of a poor performance, its future adaptations to different virtualization system would have been complicated or even impossible.

3.2.2 Component-based Programming

The component-based paradigm [255, 272] of software development enables applications' developers to construct complex large-scale systems, which can be assembled of heterogeneous functional or logical components (with diverse functionality, used programming language, etc.) and distributed across the network. The components provide well-defined processing functionality and interfaces, using which they communi-

cate with each other. Thus, once the components are composed into a larger system, they are able to realize the requested functionality.

Although it is obviously possible to encapsulate the whole requested functionality into a single component, the creation of several components cooperating on the requested processing provides several important advantages:

- *Programming simplicity and flexibility* – once the complex functionality is divided into several cooperating components, each of them might be independently developed. Thus, instead of developing a single complicated application, one can focus just on the functionality, that the particular components should provide, and even choose the development system/platform, which suits best for their implementation.

  Moreover, if a misbehavior appears in a particular component, it might be modified and changed without any affects on the other ones.

- *Components' reusability* – specialized components (e.g., compression components, encryption components, data distributing components[7], video-transcoding components, RTP streams synchronization components, etc.) can be reused across many applications. This allows faster delivery of requested functionality and reduces the time necessary for new applications' development.

- *Better resource attainability and load-balancing* – a single-component application has to be run on a single physical node, which limits the computing resources available for it. However, in the case of the same application consisting of multiple components, each component can be deployed on a different physical node, which can provide better load-balancing and resource management capabilities (each component requests a portion of the whole computing resources required for the particular application).

- *Simpler performance bottlenecks' identification* – performance bottlenecks can be identified, and the needs for performance improvements can be localized in a small number of performance-critical components instead of searching for the bottlenecks in the whole complex application. The components can be internally optimized and/or moved between platforms to improve their performance without affecting the functionality of the whole application.

The component/module-based programming principles have already been studied and successfully used in several existing active/programmable nodes architectures. For example, the NetScript project (see the Section 2.1.2.4) has proposed a collection of Virtual Network Engines (processing elements) interconnected by Virtual Links (data paths). Moreover, the Click (the Section 2.1.2.5) and the SwitchWare (the Section 2.1.2.6) routers allow to compose the processing functionality from a set of processing modules. However, all the systems require the components/modules to be designed as objects for the particular programming language (the NetScript programming language or the C++ programming language in the case of the Click router), or to be programmed using a particular programming language (the CAML in the case of SwitchWare). Moreover, the SwitchWare allows just a sequence-based data flows among the modules and, except the

[7] By the data distributing components we mean the components, that are necessary for parallel processing of the APs (for further information on the DiProNN's parallel processing see the Section 7.4.1). These components perform a distribution of an incoming data stream over multiple, simultaneously running parallel instances, e.g., in a round-robin fashion.

Netscript, the systems allow the composed application to run just on a single physical node without any possibilities to distribute the processing load of the components across a distributed infrastructure.

Because of the presented benefits, we have decided to employ a component-based programming model for the proposed node as well. The model should consider the virtualization employed in the node and should allow applications' composition in the simplest and most comfortable way as possible (the feature *F3*). Thus, similarly to the existing ones, the employed programming model should enable the DiProNN users to compose the required complex functionality from several, single-purpose simple active programs and communication channels among them defined (such an association will be further denoted by the term *(DiProNN) Session*). However, since the designed node should enable its users to develop the APs for arbitrary execution environments, the model has to take into account the APs encapsulated in their own VMs as well.

3.2.3 Possibilities of Parallel/Distributed Processing

Since many applications, which the active networks can be advantageously used for, require a real-time stream-based processing, the proposed node should allow creation of a processing environment, which provides as low latency of data processing as possible. For example, a processing of multimedia streams for collaborative environments require such a powerful system—even thought the human's visual perception does not register as low latencies as the perception of the sound[8], for truly collaborative environment the video must be precisely synchronized with the audio (so-called "lip synchronization"), and thus the video processing latency becomes of the same importance [126]. Another example are applications processing data for the haptic interactions, where the force computing algorithms generally require high sampling rates (typically, 1 KHz), and thus the processing latency has to be as low as 1 ms [116].

Since such processings are usually fairly complex (for example, the processing of video streams often involves multiple discrete cosine transformations and complicated matrix transforms, and other computationally intensive operations), a distribution of the processing load across several processing elements (e.g., processors) is necessary. However, such a distribution is not always sufficient, e.g., when high amounts of data have to be processed. Thus, to avoid bottlenecks formed by the networking interface of a single computer, the distribution of the processing load has to be combined with a distribution of the network load as well (by distributing the network load across multiple processing nodes).

There have been proposed several architectures, which address these issues. For example, the active/programmable ones presented in the Section 2.1.4—whilst the CLARA and the Cluster-based AR focus on the multimedia streams processings only, performing the data distribution and parallel processings on a cluster of commodity PCs, the LARA and LARA++ architectures allow processings of an arbitrary data streams on a dedicated distributed infrastructure based on the Cerberus hardware [55]. However, both the LARA and LARA++ use such a distributed infrastructure just for a distribution of the APs, not for their simultaneous parallel processing.

Another example could be the non-programmable processing infrastructure presented in [126] (called *Distributed Active Element*), which is used for a distributed processing and delivery of multimedia streams in a collaborative environment. As a part of this work, the author also proposes the *Fast Circulating Token* algorithm [126], which becomes

[8] In general communication, the human is able to register as low sound latencies as 100 ms.

very useful in situations, when one wants to decrease an unwanted packet reordering introduced by a distributed processing. And there are many other architectures—e.g., the *Borealis* [1], the *Chromium* [129], or the *Eclipse* [231]—that perform a distributed processing of data streams as well.

The common characteristic of all the mentioned architectures[9], which perform a distribution of the incoming data stream over several processing nodes providing the requested processing in parallel, is, that they consist of a distributing node, several processing nodes, and usually an aggregating node. The distributing node distributes the incoming data over the processing nodes, which provide the required computations, whilst the aggregating node aggregates the processed data in some a way (e.g., performs proper packet ordering). One can notice, that such a processing infrastructure could be easily defined using the component-based programming facilities, which the proposed DiProNN node provides.

Nevertheless, to make the proposed node capable of providing sufficient computing power for such a parallel/distributed processing, its architecture has to comprise several processing nodes interconnected with a powerful communication infrastructure. On the one hand, the node should enable simultaneous processing of multiple instances of a single AP, over which the incoming data are distributed by a special data distributing component (this processing will be further denoted as the *parallel processing*). And moreover, on the other, the node should also allow a partitioning of the whole processing application, which consists of multiple components, over a network of multiple nodes in the way, that the selected nodes cooperate on the whole session processing (this processing will be further denoted as the *distributed processing*). (Both these processings cover the feature *F4*).

Last, but not least, because of synchronization purposes (e.g., to allow the proper functionality of the mentioned Fast Circulating Token algorithm, when a proper packet ordering is required), the node should be able to provide a low-latency communication among the parallel instances spread over the processing nodes (the feature *F7*).

3.2.4 Fine-grained Resource Management System

Real-time network applications, which need to achieve a particular end-to-end performance, usually make use of resource[10] reservations. The applications usually specify quality of service (QoS) requirements during an establishment of the network connection, and the QoS system in turn guarantees, that (modulo system failures or preemptions) the reservation will not be reduced during the lifetime of the application. Otherwise, when the applications do not have access to the requested resources in time, the end user could notice a glitch or drop in the presentation quality. [80, 283]

For the Internet, several concepts for implementing the QoS at the IP level have been proposed—in particular, the *Integrated services (IntServ)* and the *Differentiated services (DiffServ)*. The IntServ approach is based on signalling the QoS requests from application to the network (using the *Resource Reservation Protocol (RSVP)* [36]) and allocating all the resources on the intermediate routers on the path from the sender to the destination. Due to the scalability problems of the RSVP [275] (the inner network nodes have to maintain a per-flow state), the more scalable DiffServ approach has been proposed. The DiffServ

[9] Not including the LARA and LARA++ architectures, which just distribute the computing programs across several processing nodes.

[10] "A resource" is a certain capability of the node (processing power, data storage, etc.) that is offered at a certain amount for a specific consumer.

is based on classifying the traffic, when it is entering the network, on boundary nodes. Once classified, the traffic is attributed to different behavior classes, which are provided by different processing priorities on the inner nodes.

In comparison with traditional networks, there are more resources shared among the users of the active/programmable nodes—the CPU cycles, state storage capacity, and data storage together with traditional networking components like packet queues on network interfaces. Since a precise specification of the requested resources is needed, and since the active/programmable networks do not aim to support a huge number of users (but mainly focus on a specialized user groups requesting additional services from the network), the IntServ-based approach becomes more suitable for them.

There are many works [2, 88, 169, 202, 216, 256, 282] making use of the resource allocations/reservations for real-time or time-sensitive applications (usually multimedia ones) for the traditional networks. In the active/programmable networks area, the QoS issues have also been more or less taken into account (PLAN(et), Active Network Architecture, ANN—see the Section 2.1), however, these were usually not designed and/or implemented in details. Thus, the active/programmable node architecture, which provides the most comprehensive study and design of the resource management and QoS support, is the FAIN active node (see the Section 2.1.2.2). The FAIN project defines a resource control framework that partitions and allocates the available resources (including computing resources such as CPU time and memory, and network resources such as bandwidth and routing table), and which is able to provide both *hard allocations* (the application has guaranteed access to the whole allocated capacity, even if there is a congestion in the node) and *soft allocations* (the application is guaranteed to receive the requested resources, but only if there is no congestion in the node).

As discussed in the previous section, the time-sensitive applications should be also supported by the proposed node. Thus, once the node becomes used in a shared network environment, important challenges in the design of its resource management system (RMS) have to be taken into account. On the one hand, the node should allow its users to specify resources required for proper run of their applications, and on the other, it should allocate and guarantee the requested resources during the applications' lifetime (under all circumstances—the *hard* allocations). Since the DiProNN applications can consist of several components (active programs), the users should be allowed to define the amount of requested resources for individual components—the feature *F5*. The kinds of resources, that we consider sufficient for such a programmable node, are the following:

- **CPU time** – the time required for an execution of an active application,
- **Amount of free memory** – amount of a short-term storage (e.g., RAM) necessary for proper active applications' execution,
- **Data bus bandwidth** – applications' capability for data transmissions performed through a data bus being used for data exchanges among various physical components of inside a single element and/or the whole node,
- **Amount of free storage capacity** — amount of a free space in physical devices being used for storing data for long periods of time (hard disk, shared storage, etc.),
- **Input bandwidth of a network** – maximum rate, at which the particular active application can handle incoming data from the network interface,
- **Input network buffers** – used for queueing the incoming packets before their transmission to another elements of the node,

- **Output bandwidth of a network** – maximum rate at which the particular active application can transmit outgoing data to the particular network interface,
- **Output network buffers** – similarly to the input network buffers, the output network buffers are used for queueing the outgoing packets after they have been handled from the processing applications.

For the security and accounting purposes, the running active applications have to be properly isolated from each other in the sense of a resource sharing (the feature *F6*). A malicious user or a compromised application should not affect another applications sharing the same physical resource nor it should be able to affect the resource itself. Such a strong isolation is very important to eliminate a hidden influence among the applications (e.g., through a swapping of virtual memory pages) and is the basis for resource monitoring and accounting capabilities (once the applications are properly isolated, it is quite easy to identify, what resources and for how long the particular application was consuming).

3.2.5 Flexible Data Transmission Protocol Architecture

The common task of all the transport protocols is to transmit data between the sender side and the receiver side. However, the requirements on the transfer quality (e.g., reliable communication, in-order vs. out-of-order delivery, flow and/or congestion control, etc.) differ vastly depending on the application they are used for. In general, the transport protocols could be divided into two groups: the *pure transport protocols*, which operate directly on top of the Network layer of the ISO/OSI Network model [288] (usually, on top of the IP protocol [221]), and the *application-level transport protocols*, which make use of an underlying pure transport protocol (usually, the UDP).

Besides the mostly known pure transport protocols—i.e., the *User Datagram Protocol (UDP)* [220], the *Datagram Congestion Control Protocol (DCCP)* [157], the *Transmission Control Protocol (TCP)* [131], all of which are described in the Chapter 5—there is a variety of specialized lesser-known protocols, some of which should be mentioned as well. For example, the *NETwork BLock Transfer Protocol (NETBLT)* [65], which is intended for the rapid transfer of a large quantity of data between computers, the *Versatile Message Transaction Protocol (VMTP)* [59], which is a transport protocol designed to support remote procedure call (RPC) and general transaction-oriented communication, or a set of customized transport protocols provided by the *Cactus* [40] or *Horus* [227] systems, which allow building customized protocols from collections of fine-grained specialized modules.

Examples of application-level transport protocols then include the *Real-time Transport Protocol (RTP)* [236], which provides end-to-end transport functions suitable for applications transmitting real-time data, the *Reliable Multicast Transport Protocol (RMTP)* [209] providing lossless delivery of bulks of data from one sender to a group of receivers, the *UDP-based Data Transfer Protocol (UDT)* [107], which is a high performance data transfer protocol designed for transferring large volumes of data over high speed wide area networks, and many others (e.g., the FRTP [287], the LambdaStream [285], etc.). All these application-level transport protocols make use of the underlying UDP pure transport protocol.

As we have already depicted during their introduction, the basic idea behind the active/programmable networks is to allow a simple and dynamic deployment of new services. Since such services might want to use unusual or novel transport protocols, the architectures presented so far support such a functionality by operating just on top of the

Network layer of the ISO/OSI Network model and locating all their control information just behind the packet's IP header. Similarly, the usage of arbitrary transport protocols should be supported by the proposed node as well—its communication layer must not force the users to use a particular transmission protocol, so that they should be able to use an arbitrary transmission protocol depending on the nature of their passing data.

3.3 Comparison with Existing Approaches

The previous chapter indicates, that even though we have presented just the most pioneering works in the active/programmable networks area, there have been proposed many various active/programmable routers'/nodes' architectures. In spite of this fact, as far as we know, in the community there have not been any attempts to study the features, which the virtualization principles can provide to them, and which would have been provided by any existing architecture[11]. The proposed architectures usually cope with the fundamental issues of the ANs—the execution environment and programming flexibility, the resource isolation and security, etc.—by employing their own proprietary solutions and/or operating systems' (NodeOSs') facilities. As we have depicted so far, we claim, that these issues can be easily and essentially addressed with the use of the virtualization, which can further provide another useful benefits as well.

Even thought not professing to the active/programmable networks' area, there has been one VM-based architecture mentioned in the previous chapter, which follows their basic principles. That is the node proposed by the XenBEE project (the Section 2.2.4.7), whose basic idea is very similar to the one of our work. The XenBEE also allows a server application (e.g., an AP), which could be encapsulated in a user VM running an arbitrary supported EE to be deployed on the host VM-aware system. And similarly to the DiProNN's architecture described in the following chapter, besides the main control daemon, which controls the particular node, the XenBEE also employs per-VM daemons running in every uploaded virtual machine, which communicate with the main control daemon and start the applications inside the relevant VM depending on its instructions. Nevertheless, in comparison with the XenBEE, the architecture we propose is more complex and provides more useful features, which the XenBEE is not able to provide—see the discussion later in this section.

Anyway, the virtualization principles have been also successfully used in various non-programmable network architectures, which are described in the previous chapter—for example, the Grid and Cloud computing systems, the PlanetLab and XenoServer infrastructures, etc. The common characteristic of these architectures (let us denote them as *computing systems*) is, that all of them are able to provide an access to virtual computers (in fact, virtual machines running certain OSs) located (from the users' point of view) "somewhere" in the network. These VMs usually run on a powerful infrastructure and the end users can use them for various massive computations in the same way as they do on a common personal computer (i.e., via a remote login and common OS's functions). From a particular point of view, these computing systems could be also considered very similar to our work—the proposed DiProNN node also allows remote login possibilities (even thought this is not the primary goal) and thus allows to perform user-controlled on-

[11] Several architectures use the process-level virtual machines, namely the Java VMs, to ease the portability of active programs distributed over the active network (ANTS, FAIN, OPENET, etc.). There are also architectures assuming system-level virtual machines created on top of their proprietary abstraction levels (for example, the SANE architecture in the SwitchWare project described in the Section 2.1.2.6), however, none of them tries to utilize VMs' benefits in the depth we do (i.e., the (whole) platform-level virtualization).

line computations inside the network. However, the idea behind such systems is essentially different in comparison with the proposed DiProNN node. Whilst these computing systems focus on providing virtual computers behaving as common PCs with desired performance, the DiProNN focuses on the programming capabilities, which can enable the users to flexibly and comfortably define the processing performed over their passing data. Thus, based on this specialization, the DiProNN node (and more generally, the network of the DiProNN nodes) offers several features not (directly) available in these architectures, i.e.:

- *It allows user programs to run on top of specialized EEs* – besides the XenBEE, the other computing systems are not able to run (active) programs in specialized execution environments, possibly provided by the end users. As we have depicted before, specialized EEs can improve the performance of some applications, since they have to neither provide support for legacy interfaces (as the general-purpose OSs have to provide) nor remote login capabilities required for accessing the virtual computers.

- *It allows comfortable component-based programming and automatic components deployment* – in spite of the fact, that the applications' components could be also deployed over several virtual computers provided by the mentioned computing systems, the whole deployment and data flows' setting process has to be performed *manually* in comparison with DiProNN, where this process is fully automated.

- *It allows tightly-coupled parallel processing* – even thought the parallel AP instances could be manually spread over several virtual computers as well, these computers could be dispersed over distant physical nodes[12], which can make such a processing inefficient (especially, in cases when a latter ordered composition of the instances' outputs is needed, since the data could pass throught different paths of the public networking infrastructure, which can result in different end-to-end latencies).

- *It provides high-throughput inner data networking infrastructure* – the mentioned computing systems cannot cope with the situation, when the inner processing components generate high amounts of data (even thought the application's inputs and outputs request low bandwidth only)[13]—the components, which are dispersed over (possibly distant) physical nodes, have to communicate over the public networking infrastructure, which need not provide sufficient throughput, and which increases the overall communication latency as well.

- *It enables low-latency control communication among the APs* – since the components might want to communicate with each other as fast as possible, the latency provided by the public networking infrastructure could be also too high for fast components' synchronization, state sharing, etc.

[12]The users are not aware of the physical location of their virtual computers—the computing systems provide just an access to the virtual computers providing requested computing power.

[13]For example, the real-time computations for haptic interactions as presented in the Section 10.4—based on a current haptic device position, the computing nodes have to perform massive computations of deformations and forces, which generate high amounts of data, that are later filtered and delivered to the client(s).

Chapter 4

DiProNN: Distributed Programmable Network Node

In this chapter, we present the architecture of the proposed *Distributed Programmable Network Node (DiProNN)*, which illustrates the discussed benefits of employing the virtualization in the active/programmable networks area, and which satisfies the objectives stated in the previous chapter. The proposed architecture is presented in the most general scenario, however, since such a general architecture is not always necessary, the Section 4.7 describes its available modifications leading to the ease of its application.

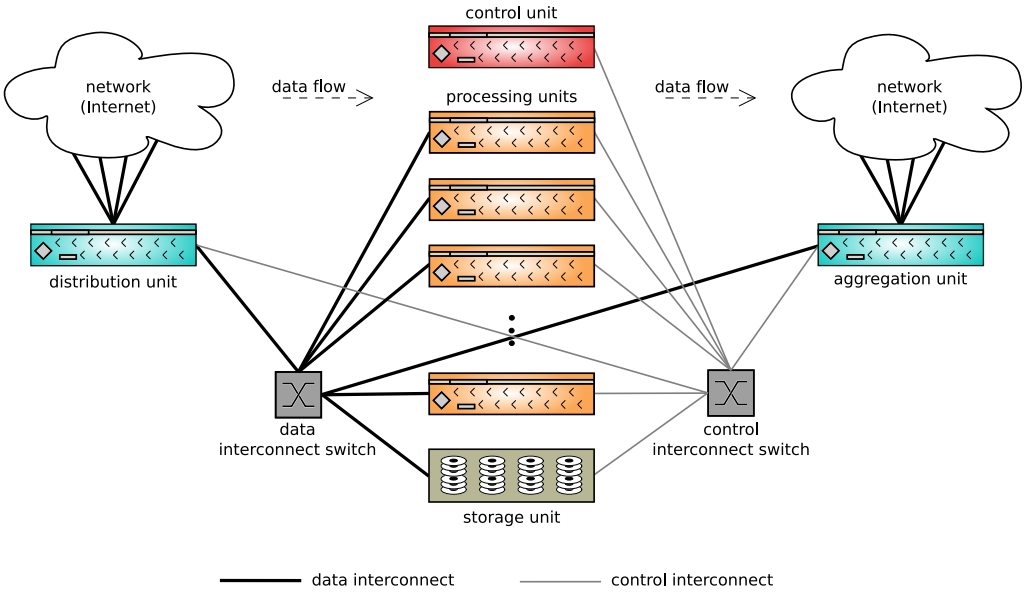


Figure 4.1: Model architecture for implementing the DiProNN.

The DiProNN's architecture assumes the infrastructure depicted in the Figure 4.1. The DiProNN units form a computer cluster consisting of commodity PCs, that are interconnected with two interconnections—a *control interconnection* used for an internal communication inside the DiProNN, and a *data interconnection* used for user data transmissions (for details about both the interconnections see the Section 4.6). Such a parallelized architecture makes the DiProNN being capable of processing higher amounts of data, since it allows the distribution of both the processing and network loads over the Processing

Units. The number of Processing Units is arbitrary, thus, the node should be able to satisfy most applications' bandwidth/processing requirements (even the higher ones).

From the high-level perspective of operation, the incoming data are first received by the Distribution Unit, where they are forwarded to appropriate Processing Units for processing. Once being processed, the data are finally aggregated using the Aggregation Unit and sent over the network to the next DiProNN node (or to the receiver). As obvious from the Figure 4.1, the DiProNN's architecture comprises five major parts, that are further described in detail in the following sections:

- **Distribution Unit**—the Distribution Unit takes care of forwarding the ingress data flows to appropriate virtual machines running on the Processing Units; the forwarding is determined by the Control Unit,
- **Processing Units**—the Processing Units receive incoming packets and forward them to the proper active programs for processing. The processed data are then forwarded to next APs for further processing, or to the Aggregation Unit to be sent away.
- **Control Unit**—the Control Unit is responsible for the whole DiProNN node management and communication with its neighborhood (including the communication with users to negotiate new DiProNN Sessions' establishments and, if requested, providing feedback about their behavior).
- **Aggregation Unit**—the Aggregation Unit aggregates the processed traffic to the output network line(s).
- **Storage Unit**—the Storage Unit serves as a DiProNN service providing the built-in APs and the built-in virtual machines, which serve as the EEs for the standalone APs, or which provide a specialized functionality. If necessary, it might also serve as a storage for users' data produced during the processing.

All the DiProNN units as well as virtual machines running on them operate on a private network segment, where each VM/unit has (and is referenced by) its own unique IP address. This ensures easier VMs'/units' addressing without any interventions with other network devices, as it may happen in cases of sharing a single network segment.

4.1 Distribution Unit

As already depicted, the Distribution Unit receives incoming users' data and forwards them to appropriate virtual machines, which run on the Processing Units, for the processing. The forwarding is controlled by the Control Unit, which sets it during sessions' establishments.

In order to enable remote node management as well as in order to enable the users to ask the node for relevant information and new sessions, the Distribution Unit listens on a well-known *DiProNN Control Port*, where all the node-related messages/requests are delivered (further details about the remote node management are provided in the Section 5.2.2). Once received, the messages/requests are forwarded to the Control Unit, where they are processed by an appropriate node-related module.

Similarly, as soon as a new session is established, the Distribution Unit maintains its private *Session Control Port* (specified during the establishment process), where the users are able to control the processing and request relevant information about its actual state.

The messages, that have been delivered to the Session Control Port, are also forwarded to an appropriate session-related module running on the Control Unit (see later).

Last, but not least, the Distribution Unit runs at least two modules—the **Control module**, which communicates with the Control Unit and controls the behavior of the unit (especially, the forwarding rules), and the **Resource Management module**, which provides relevant information about the unit's state, utilization, and available resources. If requested by the Control Unit, the Resource Management module should be also able to allocate unit's resources.

4.2 Processing Units

Since the Processing Units could be seen as independent programmable network nodes[1], their architecture is based on the generic model of an active router with loadable functionality proposed in [122]. Thanks to its modular architecture, we have modified the scheme in order to make the node being able to provide all the features we requested.

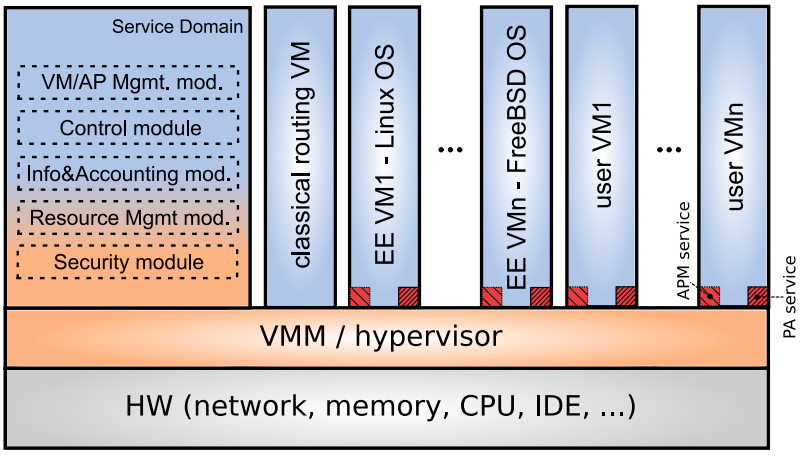


Figure 4.2: DiProNN Processing Units' Architecture.

The Processing Units' architecture is shown in the Figure 4.2. To make the DiProNN node capable of hosting virtual machines, inside of which the active programs processing the user data run, these units (and only these) are required to run a virtualization system. In spite of the fact, that the architecture, which is being described, assumes the platform-level virtualization systems, the DiProNN can be implemented using other virtualization systems as well (see the Chapter 11).

The Service Domain[2], which runs on every Processing Unit, has to manage the whole unit's functionality including uploading, starting, stopping, and destroying the VMs, the communication with the Control and Storage Units, and the sessions' accounting and management. To provide all of these, the Service Domain has to run at least the following set of modules:

- **VM/AP Management module** – the VM/AP Management module has to control all the virtual machines running on the particular Processing Unit, and the APs

[1]The Processing Unit uploads active programs and/or virtual machines, and processes them over passing user data in a similar way, as common programmable nodes do.

[2]The *Service Domain* is a specialized VM/console, which controls the particular virtualization system.

running inside of them. The module manages two kinds of the VMs: in the case of built-in VMs providing a specialized functionality (for example, the VM providing classical routing as shown in the Figure 4.2), the module is asked by the DiProNN Control & Sessions Management module for obtaining the particular VM's image from the Storage Unit. Once obtained, the particular VM is started by the VM/AP Management module without another interventions.

The other kind—the sessions' VMs—are represented by the VMs uploaded by the users and the built-in VMs, which serve as EEs for the standalone APs (both the built-in and uploaded ones). In this case, once the VMs are started, the communication with the Active Program Manager service (see later in this section) follows. The goal of this communication is to upload (just in the case of built-in VMs serving as EEs) and start the APs required for sessions' processing.

To summarize, the VM/AP Management module has to provide at least the following functionality:

- VMs' uploading, starting, stopping, suspending (making them non-active—saving their current state and freeing previously allocated resources), resuming (re-activating a previously suspended VM), migrating (moving a VM to another DiProNN's Processing Unit), and destroying (cleaning),
- communication with the DiProNN Control & Sessions Management module of the DiProNN's Control Unit in order to receive built-in VMs providing specialized functionality, and to control them (start, stop, suspend, resume, migrate, etc.)
- communication with the Control module of the Storage Unit in order to obtain necessary built-in APs/VMs,
- communication with the DiProNN Session Operator module of the DiProNN's Control Unit to obtain the sessions' description and VMs, to manage the number of AP's parallel instances running[3], to receive information about necessary VMs' migrations, to return the particular VMs during the process of session's termination, etc.,
- in cooperation with the Active Program Manager service uploading, starting, and stopping, the active programs of sessions' VMs.

- **Control module** – the Control module has to control the Processing Unit's behavior and perform the functions requested by the Control Unit. It has to provide at least the following functionality:
  - communication with the Control Unit (with the DiProNN Control & Sessions Management module) in order to register the particular Processing Unit and respond to liveness requests, to receive information about new DiProNN Sessions' establishments/terminations, etc.
  - communication with the Port Associator service (see later in this section) of sessions' VMs, which run on the particular Processing Unit—this is necessary for the registration of communication interfaces, as we describe later,

[3] As we have depicted during the motivation chapter, the DiProNN is able to process the APs in parallel. In the Section 7.4.1, we describe this functionality in detail, including the instances' adaptations to the amount of data processed.

  - providing information about the registered communication interfaces to the DiProNN Session Operator module, so that it is able to create a set of forwarding rules for ensuring the proper data flows among the session's APs (the forwarding rules and DiProNN data flows are further discussed in the Chapters 6 and 7),
  - setting and removing the forwarding rules, that have been created/withdrawn by the DiProNN Session Operator module.

- **Information & Accounting module** – the Information & Accounting module serves as the unit's information service, which both logs and provides information about actual state and various events, that have occurred during the units' runtime. The covered events include the events related to the particular Processing Unit as a whole as well as the events related to the VMs/APs actually running—e.g., starting/stopping the unit, uploading/starting/destroying VMs/APs, suspending/resuming VMs providing special functionality, performed VMs' migrations, etc. All the unit's events are logged, and if requested, delivered to the Control Unit (and later provided to the node administrators/users).

  Besides the logging purposes, if desired, this module could be also used for accounting purposes. In cooperation with the unit's Resource Management module, it may log the real session's usage of the relevant Processing Unit's resources—the acquired statistics might then be used for charging the DiProNN users for their resources' consumptions.

- **Resource Management module** – this module has to monitor and control the processing resources of the particular Processing Unit. The Resource Management module should at least:

  - monitor the actual usage of all the intended resources, and in cooperation with the Information & Accounting module, log all the acquired information for accounting purposes,
  - in cooperation with the DiProNN Control & Sessions Management module or the DiProNN Session Operator module, allocate the resources required by the particular VMs,
  - communicate with the Control Unit (its DiProNN Resource Management module) to provide information about the available Processing Unit's resources and their actual usage.

- **Security module** – the Security module monitors the behavior of the running VMs (e.g., their network communication). If a misbehavior is detected, it is logged in cooperation with the Information & Accounting module, and in cooperation with the DiProNN Control & Sessions Management module or the DiProNN Session Operator module, solved depending on the security policy defined (e.g., the misbehaving VM is immediately killed, while subsequently, the whole DiProNN Session, which the misbehaving VM belongs to, is stopped). Finally, through the Control Unit, the user is informed about the incident.

  If the particular DiProNN implementation allows more standalone APs to be run in a single built-in VM serving as an execution environment, the Security module cooperates with the APM service of the relevant VMs, which performs the monitoring of the APs running inside.

Besides the set of mentioned modules, which has to run inside of the Service Domain of every Processing Unit, there are two services, that has to be run inside of all the sessions' VMs—the *Active Program Manager service* and the *Port Associator service*. The functionality of both the services is as follows:

- **Port Associator service (PA service)** – the PA service serves for an association of the communication interfaces with real port numbers (as depicted before). The service monitors the free network ports of the particular VM and, if requested by an active program, performs the association of the particular communication interface (defined by its name) with chosen free network port. The information about the associated couple is then sent to the Control module of the Processing Unit, which the VM is running on.

- **Active Program Manager service (APM service)** – in the case of sessions' VMs, that have been provided by the users, the APM service manages the included users' active programs, while in the case of built-in VMs serving as the EEs, it serves for uploading the standalone APs into them as well. Besides this, in both the cases, it starts/stops the (uploaded) APs depending on instructions provided by the VM/AP Management module of the particular Processing Unit. Once the APs are started, they contact the PA service of the relevant VM and associate their communication interfaces as mentioned before.

  Furthermore, if the particular DiProNN implementation allows more standalone APs to be run in a single VM serving as an execution environment, the APM service monitors the behavior of the APs running in the particular VM, and cooperates with the Security module of the relevant Processing Unit, when an security incident is detected.

4.3 Control Unit

The Control Unit is the main control point of the whole DiProNN node, which controls its behavior, manages information about its state and capabilities—e.g., built-in APs, available EEs, established sessions, resources available, etc.—and responds to users' requests. It communicates with all the other DiProNN units, collects all the necessary information, and decides about various events in the DiProNN (e.g., accepts/refuses new DiProNN Sessions, chooses appropriate EEs for the standalone APs, decides about VMs' migrations, starts/terminates the sessions, collects all the accounting information, etc.). Whilst the modules, that are described in the previous section, maintain the necessary information for a single Processing Unit only, the Control Unit communicates with all of them and collects all the necessary information throughout the whole DiProNN node. The collected information is then used for various decisions as well as provided to DiProNN users, if required and permitted.

As already depicted in the section introducing the Distribution Unit, all the users' requests are delivered throught it to the Control Unit—to the DiProNN Control & Sessions Management module in the case of node-related requests, or to the DiProNN Session Operator module in the case of a session-related requests—where they are processed. Especially, the new session establishment requests are delivered to the DiProNN Control & Sessions Management module, which (depending on the actual resources' utilization and/or node state) replies, whether they could be satisfied or not. If the request could be satisfied by the particular node itself (the one which has received the request),

the Control Unit decides, which Processing Units will all the VMs, that belong to the particular session, run on (so-called APs/VMs mapping process—for details see the Section 7.3.1). Once this process finishes, the Control Unit informs all the relevant Processing Units' resource management modules to allocate the requested resources and starts the DiProNN Session Operator module serving the particular session, and passes it all the relevant information (especially, the APs'/VMs' mapping).

If the new session establishment request could not be satisfied by the particular node itself (e.g., there is not enough resources to satisfy the session's requirements), the Control Unit contacts the Control Units of all DiProNN nodes available in the network and asks them for participating on the session's processing (see the Section 7.3). If all the polled nodes do not have sufficient resources to satisfy the session's requirements, the establishment request is refused.

To provide all of the described functionality, the Control Unit has to run the following set of modules:

- **DiProNN Control & Sessions Management module** – the DiProNN Control & Sessions Management module is dedicated for controlling the behavior of the particular DiProNN node, communication with its neighborhood (another DiProNN nodes in the network), and managing all the established DiProNN Sessions as well as incoming establishment requests. The module has to provide at least the following functionality:
  - performing DiProNN units' registrations, checking units' liveness,
  - processing and responding to users' node-related requests—especially, receiving new session establishment requests, and in cooperation with both the DiProNN Resource Management module and the DiProNN Access Management module of the particular node, or in cooperation with other DiProNN nodes in the network, accepting or refusing the requests,
  - in cooperation with the DiProNN Resource Management and DiProNN Information & Accounting modules decides, which built-in EEs all the standalone APs will run in (depending on their EE requirements), and which Processing Units will then all the sessions' VMs run on (the APs/VMs mapping process),
  - communication with the Distribution Unit, e.g., to establish the control ports for new sessions, etc.
  - managing the DiProNN's specialized functionality and setting necessary forwarding rules for their proper operation,
  - in cooperation with the DiProNN Resource Management module makes decisions, which virtual machines should be migrated because of an effective usage of the resources (in the case of migrating a session's VM, this information is further provided to the particular DiProNN Session Operator module, which performs the migration itself),
  - in cooperation with the Security modules of all the Processing Units processes the security incidents,
  - communication with all the other Control Unit's modules to receive/provide necessary information (e.g., list of built-in APs/EEs, node access policy, resource usage, security incidents, etc.)

- **DiProNN Session Operator module** – the DiProNN Session Operator module controls the behavior of a particular DiProNN Session. Once the session is accepted, the module receives the information about the APs'/VMs' mapping from the DiProNN Control & Sessions Management module, and invites the user for uploading the user APs/VMs. Depending on the mapping, the module forwards the particular APs/VMs to the VM/AP Management modules of appropriate Processing Units, which are also informed about the built-in APs/VMs, that should be obtained from the Storage Unit. Once the uploading finishes, the VM/AP Management modules are further provided with an information, which APs (and thus VMs) should be started. As soon as all the APs are started and do have their communication interfaces registered, the DiProNN Session Operator module creates a set of forwarding rules, that are later applied on the relevant units to ensure proper communication channels.

  Once the session is started, the module processes and responds to user's session-related requests. If necessary, it also processes all the session's VMs migrations, as requested by the DiProNN Control & Sessions Management module.

- **DiProNN Information & Accounting module** – this module maintains necessary information about the whole DiProNN node—it communicates with all the other units/modules and collects information about actual DiProNN state (e.g., established sessions, built-in APs/EEs, resources' utilization, etc.) various node events, and/or information necessary for node usage accounting. The module serves as the main information service for the DiProNN users (if the node security policy allows them to receive the particular piece of information).

- **DiProNN Resource Management module** – the DiProNN Resource Management module maintains information about the actual state of intended DiProNN's processing resources. It communicates with the Resource Management modules of all the DiProNN units and collects the information about the actual state and usage of their resources. This information is used especially by the DiProNN Control & Sessions Management module for decisions about accepting or refusing new DiProNN Sessions, for the APs/VMs mapping process, and for decisions about necessary VMs' migrations in order to use the resources in an efficient way.

- **DiProNN Access Management module** – this module maintains the DiProNN's access policy used. If required, the users have to authenticate themselves through the DiProNN Control & Sessions Management module (in the case of node-related requests) or through the DiProNN Session Operator module (in the case of session-related requests) by the DiProNN Access Management module before they can ask the node for processing the request (e.g., the new DiProNN Session request). The authentication could be performed, for example, using a shared secret knowledge, registered username and password, the public key infrastructure (PKI) authentication using X.509 certificates, etc.

  When the authentication process finishes, the users are allowed to control the DiProNN node depending on their authorization (e.g., a common user vs. the node administrator).

4.4 Aggregation Unit

The Aggregation Unit aggregates the traffic, which has been processed on the Processing Units, to outbound network line(s). Usually, the unit serves as a simple gateway between the local private network area, which the DiProNN units operates in, and the Internet. However, sometimes a more sophisticated functionality is necessary—the Aggregation Unit, which is obviously controlled by the Control Unit, may provide some final outgoing packets' modifications, e.g., network address translations (when the network packets leaving the Processing Units do not have external receiver's IP address set), necessary network packets' encapsulation (e.g., the IPSec [148] encapsulation, if the NAT traversal[4] or the VPN[5] is necessary), etc.

Similarly to the Distribution Unit, the Aggregation Unit has to also run at least two modules—the **Control module**, which communicates with the Control Unit and controls the behavior of the unit (including the forwarding rules), and the **Resource Management module**, which provides relevant information about the unit's state, utilization, and available resources. The Resource Management module should be also able to allocate unit's resources, if requested by the Control Unit.

4.5 Storage Unit

The Storage Unit maintains the DiProNN's built-in functionality—the APs, which the users might use for processing, and the VMs, which serve as execution environments for the standalone APs (the user ones and the built-in ones), or which provide a specialized functionality. The unit is mainly used by the Processing Units' VM/AP Management modules, which ask it to deliver APs/VMs/EEs, that are requested by the sessions (the delivery itself is performed using a specialized service, e.g., the FTP, SCP/SFTP, or FTPS).

Besides delivering the files/images, the unit further maintains the list of available APs/VMs/EEs, which is provided to the Control Unit's DiProNN Information & Accounting module, and later to the users. Moreover, if there is a shared network storage in the network, which provides the APs/VMs/EEs for all the DiProNN nodes, the unit has to collect them as well—once an AP/VM/EE, which is located in the network, is requested by a user, the unit has to obtain it before being provided to the requesting Processing Unit.

Optionally, the Storage Unit might also serve as a storage for users' data produced during their sessions' processing. Hence, the unit should provide a remotely-accessible network filesystem, which can be accessed by the Service Domains of all the Processing Units (e.g., the NFS [244], AFS [52], Samba [56], etc.), and which is able to allocate requested storage space for the users.

The Storage Unit should also run at least the **Control module** and the **Resource Management module**. In this case, the Control module has to maintain the list of available APs/VMs/EEs and provide the relevant files/images to the requesting modules. The Resource Management module, which has to manage and provide information about units' resources, should be further able both to allocate the requested disk space and the data interconnection's bandwidth for a particular session.

[4]The *NAT traversal* [85, 175] is a general term for techniques to establish and maintain TCP/IP network connections which traverse Network Address Translation (NAT) gateways.

[5]The *Virtual Private Networks (VPN)* [39] provide an encrypted connection between user's sites/networks distributed over a wide-area public network (e.g., the Internet).

4.6 Data and Control Interconnections

As the Figure 4.1 in the beginning of this chapter depicts, there are two independent interconnections used in the DiProNN—the data interconnection and the control interconnection. Both interconnections are served by the Service Domain of the particular DiProNN unit, and thus just the Service Domains has to support them (e.g., in the case, when they need specialized drivers/features included in the OS's kernel). The other virtual machines running on the particular unit communicate with the Service Domain via virtual interfaces provided by the particular virtualization system used—as described later in the Sections 7.3.2 and 7.4, the Service Domain then decides, which packets will be sent via the data interconnection and which ones via the control one; similarly, it also receives and forwards incoming packets to relevant virtual machines. In this section, we discuss several contemporary technologies that could be used to implement them.

The data interconnection is dedicated for data packets' (the packets that are being processed) transmissions and might be provided by common network interfaces like Gigabit Ethernet [68] or 10Gigabit Ethernet [130]. The most important characteristic of the data interconnect is the throughput it can provide, since it affects the amount of data, that could be delivered for processing (and thus the amount that could be processed, if there is a sufficient computing power on the Processing Units and/or enough Processing Units to perform parallel processing). The communication latency is less important for the data interconnections, since even being very low, the whole latency delivered by the DiProNN node will be mostly affected by the session's processing performed on the Processing Units.

The described situation is diametrically different for the control interconnection, which serves as a communication infrastructure used for control communication among the active programs running and/or the communication among the DiProNN units/modules. The available throughput, which is provided by the particular interconnect, is not so important, because the amount of control messages necessary to be transferred will be (in most cases) much lower than the amount of data messages. However, the latency provided by the control interconnect is very important, since it can affect the whole DiProNN performance—especially in the case of the parallel processing, when the parallel instances need to communicate during the data processing (to share their state, to synchronize, etc.).

Thus, the usage of a specialized low-latency interconnect providing the end-to-end latency close to a message passing between threads on a single computer, is very desirable. Nevertheless, as described in the Section 5.2.1, the network-layer and transport-layer protocols used with the particular control interconnect have to provide a transparent addressing and multiplexing features—the availability to address a particular application running on a particular host (ideally, by employing/emulating the IP network stack to allow the use of transport protocols mentioned in the Section 5.2.1). Unfortunately, this requirement could lead to an increase of the end-to-end latency for highly specialized interconnects; however, without these features, the particular interconnect could not be used in the DiProNN (see more details provided in the following chapter).

The examples of currently available and applicable control interconnects are:

- **Myrinet-2000** – the Myrinet-2000 [296] is a switched, gigabit per second proprietary network technology developed by Myricom[6]. Using so-called *GM message passing library*—a lightweight communication system designed by Myricom—the Myrinet-

[6] `http://myri.com/`

2000 is able to provide as low as 5.05 μs one-way latency for 1-byte messages [293]. However, since the GM library bypasses an operating system's IP network stack and does not provide a transparent multiplexing mechanism, it is not directly applicable in the DiProNN. Thus, an emulation of the IP network stack over the GM must be employed, which naturally increases the end-to-end latency up to about 22-25 μs for unidirectional communication and 1-byte messages as well [291].

- **InfiniBand** – the InfiniBand network [297], which is developed by the InfiniBand^{SM} Trade Association[7], provides an architecture for an inter-server communication. The InfiniBand connection's signalling operates at a data rate of 2 Gbps of effective theoretical throughput in each direction. These links can be aggregated together into groups of four (4X) or 12 (12X) capable to provide a larger capacity (4X link can pass a single stream of 8 Gbps). Moreover, the InfiniBand supports double data rates (DDR) as well as quad data rates (QDR) speeds providing link speeds up to 96 Gbps for 12X QDR connection [57].

  The InfiniBand network is able to provide as low as 4 μs one-way end-to-end latency for messages up to the size of 10 KB [123]. However, similarly to the Myrinet-2000 the InfiniBand is not directly usable in DiProNN since it uses the MPI[8] interface to transmit data. Thus, an emulation of the IP layer over the InfiniBand must be used [141], providing one-way end-to-end latency of about 17 μs for messages up to the size of 100 KB [123].

- **Quadrics** – the Quadrics Network (*QsNet*) [214] is a 2 Gbps network integrating a local virtual memory of network nodes into a distributed virtual shared memory, and providing a programmable processor in each network interface, that allows an implementation of intelligent communication protocols. The Quadrics network consists of two main building blocks—the programmable network interfaces, called *Elan*s (*Elan4*s for second-generation network mentioned later), and the high-bandwidth, low-latency communication switch, called *Elite* (*Elite4*).

  The second-generation Quadrics network, called *QsNet*^{II} [24], has reduced the one-way end-to-end latency provided by the first-generation Quadrics network from about 4.3 μs to about 2 μs (for small messages up to 32 B) [3]. While the Quadrics user-level communication libraries (the `libelan` and the `libelan4`) can provide an OS-bypass communication directly to user-level parallel programs, the kernel-level communication library provides direct support to the IP protocol, and thus is directly applicable in the DiProNN.

  Since 2006, the Quadrics also offers 10Gigabit Ethernet switches under the name *QSTenG*[9]. These products make use of experiences obtained from two generations of QsNet networks, providing the port-to-port latency of about 200 ns [305].

- **10 Gigabit Ethernet** – the technology progress, which has decreased the one-way end-to-end latency of the 10 Gigabit Ethernet as low as about 19 μs for small data packets (up to 128 B) and as low as 23 μs for data packets of about 1 KB [130], makes the 10 Gigabit Ethernet interconnect also being suitable for the DiProNN's control interconnection. The performance achieved is very good compared to the

[7] `http://www.infinibandta.org/`

[8] MPI is a language-independent communication protocol used to program parallel computers. It has become a standard for communication among processes that model a parallel program running on a distributed memory system.

[9] `http://doc.quadrics.com/`

performance of Gigabit Ethernet providing one-way end-to-end latencies of about 62 μs for 64B data packets and of about 91 μs for larger packets (approximately 1450 B) [128].

When a specialized control interconnection is not available, the usage of just a data interconnection is sufficient—the data interconnection is then used both for the data and control messages simultaneously. In this case, all the DiProNN functionalities remain unaffected, however, the control messages might become delayed not only because of a higher end-to-end latency of the data interconnection used, but also delayed depending on the interconnection's actual saturation by the data packets. Nevertheless, the saturation problem could be solved by the usage of two identical higher-latency "data" interconnections (one for the data messages and one for the control ones) and thus all the data and control packets might be transferred separately.

4.7 DiProNN's Architecture Modifications

As mentioned in the beginning of this chapter, the described DiProNN's architecture represents the most general one. The only modification of the architecture being depicted in the Figure 4.1, which has been mentioned in the previous section, was using just a single interconnection serving as the data and control one simultaneously (in the case, when a specialized (low-latency) control interconnection is not available). Nevertheless, this modification is not the only one—without any degradation of DiProNN's features[10], one can minimize the DiProNN's complexity and set up the node using fewer physical machines than the most general architecture assumes.

In the Figure 4.3, several types of the DiProNN's architecture modifications, which are based on merging the DiProNN units, are depicted. All of these modifications, which are described in the rest of this section (except the minimal one depicted in the Figure 4.3(e)), assume both the data and control interconnections. However, as discussed previously, just a single interconnection for both the data and control messages can be used as well.

The most straightforward modification is merging the Aggregation and Distribution Units, as shown in the Figure 4.3(a). In most situations, both units will not be too loaded, and thus the usage of two independent physical machines is not necessary—the merged unit then performs the functionality of both the merged ones simultaneously. Even further, thanks to the virtualization, the Aggregation or the Distribution Unit (or both) might be run in a specialized virtual machine(s) started on an appropriate Processing Unit (the one having a connection to the Internet, so that it would be able to receive users' requests). Likewise, both the units' functionalities might be performed directly by the Service Domain of an appropriate Processing Unit(s).

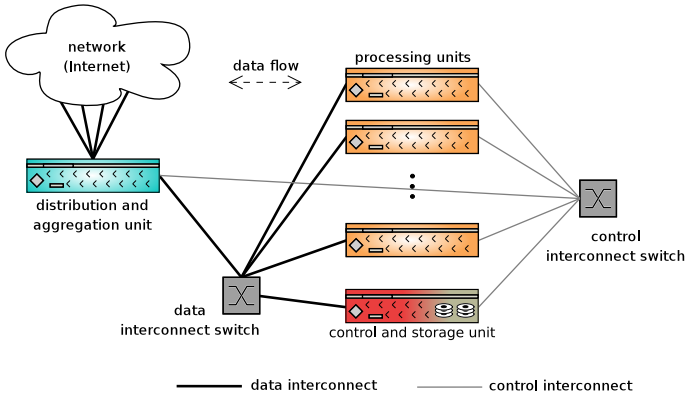
Moreover, the Figure 4.3(a) shows the Storage Unit, which is merged with the Control Unit. Nevertheless, the functionality of the Storage Unit might be also performed by a single Processing Unit (see Figures 4.3(b) and 4.3(e)), or provided by some/all of them simultaneously (the storage space is provided by some/all of them using a distributed filesystem[11], whilst one of them runs both required modules and responds to the requests)—see the Figures 4.3(c) and 4.3(d).

[10] Not including the performance feature, which, in the case of lower number of Processing Units, is obviously reduced.

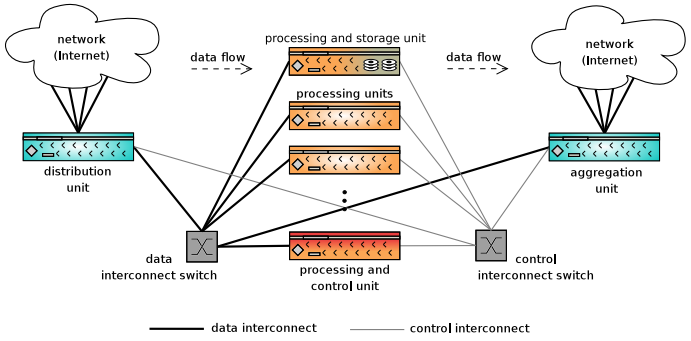
[11] For example, the AFS [52], DFS [252], Ceph [271], Lustre [239], XUFS [267], etc.

Similarly to the Aggregation and Distribution Units, the Control Unit neither requires an independent physical machine and might be thus also run in a virtual machine running on an appropriate Processing Unit(s) (the Figure 4.3(b)), performed by the Service Domain of some Processing Unit(s), or run on the same physical machine as the Distribution and Aggregation Units do (see the Figure 4.3(c)).

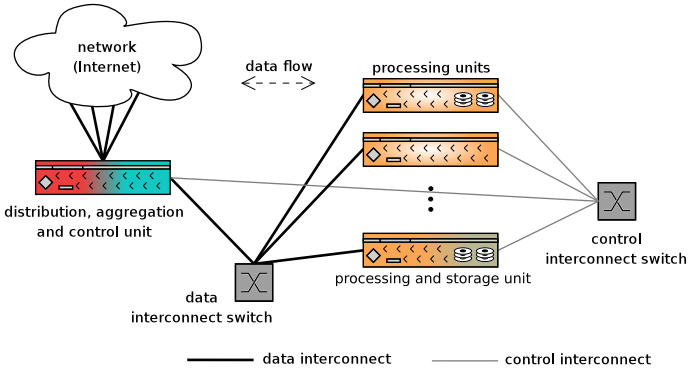
By the combination of the discussed modifications, one may get the DiProNN composed of several Processing Units only (see the Figure 4.3(d)) as well as the DiProNN's minimal architecture composed of just a single Processing Unit (see the Figure 4.3(e)). The minimal architecture provides the simplest application of the DiProNN, since there is neither data nor control interconnection necessary. As mentioned in the beginning of this section, such a DiProNN node provides all the features as the DiProNN node set up using the most general architecture described. Nevertheless, the performance of such a DiProNN node is highly limited.



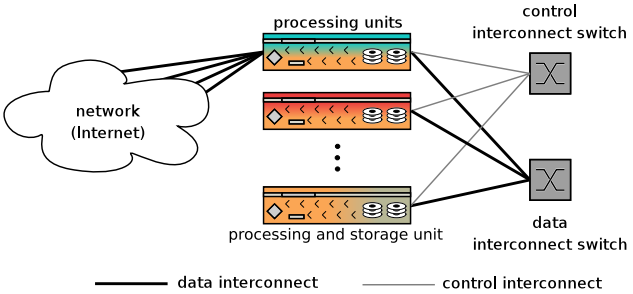
(a) Merged Distribution and Aggregation Unit. The Storage Unit performed by the Control Unit.



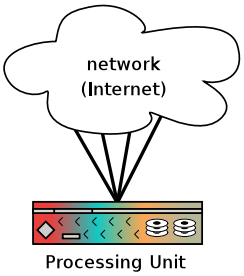
(b) Merged Control Unit with one of the Processing Units. The Storage Unit performed by a single Processing Unit.



(c) The Control Unit runs together with both the Distribution and Aggregation Units. The Storage Unit's functionality is performed by a single Processing Unit, whilst the storage space is distributed among some of them.



(d) The DiProNN composed of Processing Units only. The Storage Unit's functionality performed by a single Processing Unit, whilst the storage space is distributed among all of them.



(e) The minimal DiProNN's architecture.

Figure 4.3: Several types of DiProNN's architecture modifications.

Chapter 5

Data and Control Communication Protocols

Before we present details about both the programming model, that we propose for DiProNN programming and the DiProNN's functional specification, we describe the data and control communication protocols it uses. The data protocols are used for sessions' data packets transmissions—the packets, which contain data that should be processed—while the control communication protocols are used both for communications inside the particular DiProNN node[1] (the internal control transmission protocols) and for external DiProNN controlling and management (the *DiProNN Control Protocol (DiCP)*).

5.1 Data Transmission Protocols

As depicted in the objectives, the DiProNN should not force its users to use a particular data transmission protocol. In conformity to this requirement, it supports more of them allowing its users to choose the proper communication protocol depending on their applications' needs. And even further, the DiProNN Sessions' design (details in the Chapter 6) allows the users to use more transport protocols simultaneously throughout a single session processing.

This enables the DiProNN users to use the proper transport protocol, which suits the nature of their application and/or passing data best. For example, for applications, which require real-time transmissions with minimized delays, they can choose connection-less protocols, that do not provide any transmission quality guarantees (e.g., reliable delivery, flow control, etc.), but provide as fast delivery as possible. Against it, for non-realtime applications requiring a reliable delivery, they can use different ones. Thus, the users are also able to use different implementations of well known transmission protocols as well as their own ones—for example, different implementations of the TCP protocol described in this chapter—e.g., the TCP Tahoe [133], TCP Reno [195], TCP Vegas [37], TCP Hybla [47], etc. can be used.

The only functionality, that the DiProNN requires from such data transport protocols[2] is the data multiplexing—the protocol's ability to serve several applications and/or independent data streams for a single host (a VM in this case) simultaneously. Such a

[1] Note, that the control packets, which are encapsulated by a particular internal control transmission protocol, are transmitted over the DiProNN control interconnection described in the previous chapter.

[2] The data transport protocols' ability to make use of the underlying IP layer (or another, e.g., a currently unknown layer with similar functionalities, and/or a layer required by the type of the data interconnection used) is naturally supposed.

functionality is provided by a set of ports in the most common pure transport protocols (e.g., the UDP, the DCCP, or the TCP).

Even though slightly limiting the set of usable transport protocols, this requirement is necessary since the DiProNN has to distinguish among the data flows being sent from a VM, so that it is able to forward them to other VMs for further processing (see the session establishment process in the Section 7.3.2). Nevertheless, we believe, that this limitation is more acceptable than inserting a service/control information behind the IP header of every transmitted packet (as most active/programmable nodes do), which on the one hand can enable the usage of an arbitrary transport protocol, but which on the other wastes the available bandwidth.

The rest of this section presents four pure transport protocols[3], which satisfy the mentioned requirement and which should be supported in every DiProNN implementation. The protocols are described from the simplest one to more complex ones; in the end of this section, we conclude with a comparison of their features and kinds of applications, which they might be used for.

5.1.1 UDP (*User Datagram Protocol*)

The simplest data transmission protocol, which we describe—the UDP [220]—is one of the core protocols of the Internet protocol suite providing an unreliable communication service. It is a connection-less protocol that does just as little as a transport protocol can—aside from the multiplexing/demultiplexing function and some light error checking, it adds nothing to the underlying IP (*Internet Protocol*) [221].

UDP datagrams may arrive out of order, appear duplicated, or even go missing without any notice. Therefore, the application program, which uses the UDP, must deal directly with end-to-end communication problems that a connection-oriented protocols like, e.g., the TCP (*Transmission Control Protocol*, details in the Section 5.1.4) would have handled—for example, retransmissions for a reliable delivery, packetization and reassembly, flow control, congestion avoidance, etc.

However, since the UDP avoids checking whether every packet arrived, it is more suitable for time-sensitive applications, because the dropped packets are in cases of real-time media transmissions more preferable than the delayed ones.

5.1.2 DCCP (*Datagram Congestion Control Protocol*)

The *Datagram Congestion Control Protocol (DCCP)* (as defined in [157]) is a slightly more complex transport protocol than the UDP. It is a message-oriented transport layer protocol that implements bidirectional, unicast connections of congestion-controlled, unreliable datagrams. The DCCP is intended for the applications with timing constraints on the delivery of data, that may become useless to the receiver, if a reliable in-order delivery combined with a congestion avoidance is used. These applications, such as streaming media, can benefit from DCCP's control over the tradeoffs between the delay and the reliable in-order delivery.

The main difference between the UDP and the DCCP is the TCP-friendly congestion control mechanism provided by the DCCP. Because it is complicated to get it right, many UDP applications ignore or greatly simplify congestion control issues, even though they can lead to an application's and/or a network's misbehavior. Similarly to the UDP, the

[3] As depicted during the motivation, most application-level transport protocols operate on top of one of the pure transport protocols being described, and thus are obviously supported.

DCCP provides an unreliable flow of datagrams[4] without in-order delivery guarantees. Further, the DCCP datagrams' size is also limited by the lesser of the maximum size of an IP datagram (the underlying network's MTU, *Maximum Transmission Unit*) and the size of the DCCP datagram socket buffer. In addition to the UDP, however, the DCCP provides a path MTU discovery to determine the maximum IP datagrams' size.

5.1.3 ARTP (*Active Router Transport Protocol*)

The ARTP protocol [225], which has been designed as a part of the active router [122] developed at the Masaryk University in Brno, is a connection-oriented transport protocol providing a reliable congestion-controlled duplex communication channel without ensuring, that the data will be received in the same order as they were sent.

The data exchange between the ARTP layer and the application layer is done using the data blocks called *ARTP datagrams*, which may have arbitrary size, and thus they may not pass through the network at once. The ARTP protocol fragments them into smaller parts (called *ARTP packets*) and sends them over the network to the receiver. When the receiver receives all the fragments of a single datagram, it reassembles them into the original datagram and passes it to the receiver application. The order of passing assembled datagrams is not given—the ARTP guarantees the correct datagrams' assembling only.

Similarly to the DCCP, the ARTP also provides a congestion-control algorithm, which controls the amount of data, that is sent to the network, avoiding receiver's or network's congestion. Furthermore, the ARTP allows two types of data to be transferred—the *main data*, which are used for the data communication between the end nodes, and the out-of-band *control messages*, which are dedicated to both end-points' management. The main data can further consist of two parts—the encrypted data and its signature.

5.1.4 TCP (*Transmission Control Protocol*)

The most complex data transmission protocol, which is described in this chapter—the *Transmission Control Protocol (TCP)* [131]—is a transport layer connection-oriented byte stream protocol, which (similarly to the UDP) also belongs to the core protocols of the Internet protocol suite. The TCP provides a reliable in-order delivery of a stream of bytes, making it suitable for applications requesting an accurate delivery rather than a timely delivery—the TCP sometimes incurs relatively long delays while waiting for out-of-order messages or retransmissions of lost messages.

The TCP is stream-oriented, that is, the TCP protocol entities exchanged streams of data. Individual bytes of data are placed in memory buffers and transmitted by the TCP in *Protocol Data Units* (PDUs—also known as "segments"). Among the other features, which the TCP provides, belong an end-to-end flow control to avoid letting the sender to send data too fast for the TCP receiver (which is unable to reliably receive and process it) and a congestion control mechanism controlling the rate of data entering the network to achieve high performance and avoid a "congestion collapse", where the network performance can rapidly fall (the TCP keeps the data flow below a rate that would trigger a collapse).

As already depicted in the beginning of this chapter, there have been several versions of the TCP presented—besides the mentioned TCP Tahoe [133], TCP Reno [195], TCP Vegas [37], TCP Hybla [47], also other ones, e.g., the Fast TCP [270], CUBIC TCP [111],

[4] Both the DCCP and the UDP protocols are packet stream protocols (as opposed to, e.g., the TCP protocol, which is a byte stream protocol).

YeAH TCP [22], etc. The particular versions employ different congestion avoidance algorithms and differ especially in the way, how they react to packet losses, and how they are able to estimate (and utilize) the bandwidth available in the network.

5.1.5 DiProNN's Data Protocols Summary

As the previous sections indicate, the transmission protocols provide various features, which make each of them suitable for a specific kind of applications. To discuss the kinds of applications, which they might be advantageously used for, let us summarize all their important features, that have already been depicted, in the following table (the Table 5.1).

Feature	*UDP*	*DCCP*	*ARTP*	*TCP*
Reliable communication	✗	✗	✓	✓
In-order delivery	✗	✗	✗	✓
Flow control	✗	✗	✗	✓
Congestion control	✗	✓	✓	✓
Connection oriented	✗	✓	✓	✓
Stream type	packet	packet	packet	byte
Datagram (packet) size	restricted by MTU	restricted by MTU (MTU discovery)	arbitrary	—
Control messages	in-band	in-band	out-of-band	in-band

Table 5.1: Summary of described data transmission protocols' important features.

The main feature, that one has to decide before stream-based applications' programming, is the reliability of the communication. If the application needs to have a reliable communication guaranteed (in the sense that all the data, which had been sent by the sender, have to be received by the receiver), the user has to choose between the ARTP and the TCP protocols (and its variants). Furthermore, if the application needs to receive all the data in the proper ordering, the usage of the TCP protocol is necessary (the ARTP might be also used, but the application has to manage the proper datagrams' ordering on its own).

In situations, where the reliability of the communication is not crucial, but where the timely data delivery is highly desirable (time-sensitive applications like real-time multimedia transmissions), the usage of the unreliable UDP or DCCP protocols might become useful.

The other important feature is the type of the data stream transmission. The "packet stream" means, that all the data are packetized by the sending application and sent in data blocks of the size either limited by an underlying network's MTU (in the case of the UDP and DCCP protocols) or sent in data blocks of an arbitrary size (in the case of the ARTP protocol, where the arbitrary sized ARTP datagrams are packetized depending on the underlying network's MTU on the protocol's layer, not on the application layer by the application itself as in the cases of the UDP and DCCP). In comparison with the "byte data stream" approach, which is assumed by the TCP protocol and which means that the data are passed from the sending application to the protocol layer as a sequence of bytes, the application has better control over an independent blocks of data, that could be easily distributed and processed in parallel (further details about DiProNN's parallel processing are given in the Section 7.4.1).

Finally, in addition to the other protocols, the ARTP provides a feature, that might become useful in critical situations, and that increases the comfort of its use—the out-of-band control messages, which are prioritized from the data ones. All the other protocols require the application to send the data and control messages in-band, which might delay crucial control messages inconveniently, and to distinguish between them on its own.

5.2 Control Transmission Protocols

5.2.1 Internal Control Transmission Protocols

The set of available control transmission protocols, which could be used for internal communications via a particular control interconnection, depend on the communication model(s) it supports—since each specialized interconnection might have its own communication model(s) defined, it might require its own, special communication protocol(s). However, since the DiProNN cannot suppose, that all the VMs (including the users' specialized EEs) do support such specialized communication models, the users' APs communicate via a well-known and widely-used communication model (e.g., the IP network stack [221]) and the Service Domain (if necessary) transparently translates their communication into the communication model required by the particular control interconnection.

Generally, there are two scenarios, that might occur: the control interconnection, which is used in the particular DiProNN node, provides such a well-known and widely-used communication model, that on the one hand satisfies the addressing and multiplexing features (as discussed in the Section 4.6) and on which is directly supported by all the common operating systems, or not.

In the positive case, it can be supposed, that all the user VMs support the particular communication model, and thus the user applications are directly able to use it. The Service Domain of the particular Processing Unit then just forwards the packets coming from an application (running in the unit's VM) to their receiver (and vice versa for incoming packets)—details about forwarding the flows inside the DiProNN are provided in the Sections 7.3.2 and 7.4. An example of such a common and widely-used communication model, which will be probably used in most DiProNN implementations, is the already mentioned IP network stack, that allows a set of common transport protocols to be used above it. Since this model is also supposed for data transmissions, the applications might directly use the same transport protocols for the control messages as for the data ones—several examples of them are given in the previous section.

In the negative case—when the particular control interconnection requires a special communication model not supported in common operating systems, which is further not able to emulate a common communication model (i.e., the IP)—the Service Domain of each Processing Unit has to provide a translation[5] between the widely-used communication model, which the applications communicate with the Service Domain, and the communication model required by the particular control interconnection. This translation mechanism, however, obviously leads to increasing the communication latency, and thus must be carefully considered.

[5] As results from the objectives, the DiProNN must not force its users to include highly-special features inside their uploaded VMs.

5.2.2 DiProNN Control Protocol

The *DiProNN Control Protocol (DiCP)* serves as an interface between the DiProNN node and its users/administrators. In particular, the protocol defines a set of messages, which enable the users to remotely manage the DiProNN's/sessions' behavior and which allow them to ask the node for some information (e.g., information related to node actual state, built-in functionality, access policy, previously established sessions, etc.). Since the DiCP has not been designed in detail yet and since the particular DiProNN implementation might further want to customize the DiCP on its own (to suit its particular needs), we provide just the DiCP's basics in the rest of this chapter.

5.2.2.1 DiCP Basics

The *DiProNN Control Protocol (DiCP)* is intended to be a simple, text-based, application-level protocol used to transfer controlling and informational messages/requests to manage both the whole DiProNN node and the previously established DiProNN Sessions. Its messages should be delivered using a fully reliable transport-layer protocol[6] running over the IP network stack (or another communication model employed in the future). The concrete transport protocol, which the DiCP uses, is up to the particular DiProNN implementation; however, the usage of the previously described TCP protocol is recommended.

The requests, which are related to the whole DiProNN node management, as well as the requests for new DiProNN Sessions' establishment have to be delivered to so-called *DiProNN Control Port*—a well known port, which is established on and maintained by the DiProNN's Distribution Unit (details about serving the user requests are provided in the Section 7.2). Once the messages are received on the DiProNN Control Port, they are immediately forwarded to the DiProNN Control & Sessions Management module, where they are processed (obviously, if the user has been previously authenticated using the DiProNN Access Management module, if required).

As soon as a new DiProNN Session is established, it is assigned a new port—the *Session Control Port*—which serves for its management. The messages, which are received on this port, are thus forwarded to the DiProNN Session Operator module, which processes them and performs the required controlling/management/informational functions (obviously, if the user has been previously authenticated).

The rest of this section depicts the set of basic functions, which the DiCP has to provide. Obviously, these functions can be further enriched by the particular DiProNN implementation.

Node informational functions
The DiCP informational requests, which relate to the whole DiProNN node, have to be delivered to the DiProNN Control Port. Depending on the user's authorization, one should be able to ask the node at least for:

- *Built-in functionality* – the active programs, which the users are able to use for their processing applications (the ones available in the particular node as well as the ones downloadable from a shared data storage located in the network).

[6] By the term "fully reliable" we mean the protocol, which provides reliable in-order delivery.

- *Available execution environments* – the EEs, which the users might use for their standalone APs (again, the ones available in the particular node as well as the ones downloadable from a shared data storage located in the network).
- *Specialized functionality* – the specialized administrators' functionality (e.g., the classical routing), which the nodes performs (or is able to perform).
- *Established/Active DiProNN Sessions* – the sessions, which are running on the node. The list should contain all the established sessions together with some relevant information about them (e.g., resource usage, running time, user information, status, etc.).
- *Authentication methods used* – the list of authentication methods, which is used for users' authentication. This information should be obviously available for non-authenticated users as well.
- *Node usage/events history* – the information about various events (sessions' establishments, units' failures, security issues, etc.) and/or node usage (used resources, built-in functionality, EEs, etc.) from the history.

Node controlling functions
The DiCP controlling messages, which are related to the whole DiProNN node management, have to be delivered to the DiProNN Control Port as well. Depending on the user's authorization, one should be able to maintain/control at least:

- *Access policy* – authorized users should be able to remotely (re)define the node access policy—in particular, the method(s) used for users' authentication and users' authorizations.
- *Set of built-in APs and EEs* – in the case of a necessity to upload new built-in active programs and/or execution environments, or to remove/block the existing ones, the authorized users should be able to manage it remotely via the DiCP as well. Nevertheless, the uploading itself could be performed using a specialized service, which the user and the node agree (via the DiCP) on—e.g., the FTP, SCP/SFTP, or FTPS.
- *Sessions' terminations* – via the DiCP, authorized users have to be able to forcefully terminate an arbitrary DiProNN Session, e.g., because of security violations.
- *Units' startup/restart/shutdown* – if necessary, the authorized users have to be able to remotely start/stop/restart the DiProNN units. (e.g., if the number of Processing Units becomes insufficient/redundant).

DiProNN Session establishment
The session establishment request, which the DiCP delivers to the DiProNN Control Port, is a special node controlling request, that the authenticated and authorized DiProNN users use to ask the node for a new session. The whole establishment process comprises two parts:

- *The request delivery* – besides user's information, the DiCP message asking for a new session establishment has to contain at least so-called DiProNN Program, which defines the particular session and all the requested resources (further information about the DiProNN Programs are provided in the following chapter).

- *APs/VMs uploading* – once the session establishment request is accepted, the DiCP should provide a message announcing the uploading of the user's APs/VMs. The message is delivered to the Session Control Port, and the uploading process itself should be performed using a specialized service, which the user and the node agree on.

Session informational functions
The DiCP informational requests, which relate to a particular session, have to be delivered to the relevant Session Control Port. The authorized users, which should be defined by the session owner during the establishment process, might then ask the node for information about a particular session, e.g.:

- *Session's processing state* – the users should be able to ask for information about both the current and historical state of the session's processing (running time, amount of data processed by the particular APs, etc.)—the information should cover all the DiProNN nodes participating on the processing.
- *Session's resource consumptions* – the amount of node resources, which are/were consumed by the session's VMs/APs. Again, this information should cover all the DiProNN nodes participating on the processing.

Session controlling functions
The DiCP controlling messages, which relate to a particular session, have to be delivered to the relevant Session Control Port as well. The authorized users (defined by the session's owner) should be able to maintain/control at least:

- *Session's startup* – once the session establishment process finishes, the DiCP should provide a message asking the node for starting it (i.e., its APs/VMs).
- *Session's dynamic reconfigurations* – if required, the DiCP should provide functions, which enable the users to dynamically adapt/change the session's data flows. This can also comprise the delivery of a new session's DiProNN Program and/or new processing VMs/APs.
- *Session's termination* – once the processing finishes, the DiCP must provide a message requesting session's termination. Once the session is stopped, the DiCP should provide functions asking the node for returning particular VMs/files (the downloading itself should be performed using a specialized service).

Chapter 6

DiProNN Programming Model

In this chapter, we describe the programming model, that we propose for DiProNN programming. As depicted in the objectives, the model is based on the workflow [63] and component-based programming principles [255, 272], and was further inspired by the ideas of the StreamIt [261]—a language and compiler specifically designed for programming modern stream-processing applications.

The main goal of such a programming model is to define the processing application, that should be performed on the computing elements. Since such an application generally consists of several components, the model has to be further able to define communication channels among the components' interfaces, so that the data, that should be processed, can pass properly.

In the case of DiProNN, the components are represented by the active programs running inside virtual machines, whereas the APs' interfaces are provided by the network ports they communicate with. Thus, the DiProNN programming model has to be able to define a processing application (denoted as the *(DiProNN) Session*), which consists of a set of the APs communicating using common network services. Once such an application is defined, the components have to be deployed across the DiProNNs' Processing Units according to a precomputed schedule (the deployment scheduling is further discussed in the Chapter 8).

To ensure the communication among the deployed APs, two approaches could be used—the static one and the dynamic one. The static one means, that during the APs' startup, each of them is provided with an information, where its output interfaces should send data to (e.g., an IP address of the VM, which the receiving AP runs in, and a network port, that it listens on). This communication model does not require another interventions into the data stream, however, provides slightly limited features.

The dynamic approach means, that during the APs' startup, each of them is instructed to send data to a specific inter-mediator (e.g., to the Service Domain), which further forwards the data to the subsequent APs according to the communication channels defined. Even though one may think about introducing additional overhead because of another interventions to the data stream (which is, however, negligible in the real situations, as we show in the Section 12.2.4), this approach can provide several useful features. For example, it can enable dynamic changes of the processing application without any needs to restart the influenced APs—i.e., if a data flow outgoing the particular AP should be sent to a different AP/receiver, the inter-mediator can ensure it on its own without any needs to restart and reset the sending AP. Moreover, since the DiProNN allows a distribution of the processing application across several nodes in the network, the dynamic approach further allows inter-node migrations of the sessions' VMs (see the Section 7.4.2), which

in the case of static approach would have resulted in restarting and resetting the AP(s), which sends data to the AP(s) inside the migrated VM[1].

Even thought the combination of both approaches could be employed as well—initially, the static approach is used and, once necessary, the dynamic changes are performed[2]—just the pure dynamic approach is assumed in the rest of this thesis.

At first, this chapter elucidates the idea of DiProNN Sessions and sketches a pseudo language, which we use for the definition of their processing functionality in the *DiProNN Programs*. Later, the definition of the communication channels is presented and finally, an example of possible application, that is defined using the presented pseudo language, is sketched.

6.1 DiProNN Sessions

The DiProNN Session represents a particular processing application, which comprises of a set of active programs, a set of virtual machines, and communication channels among the APs. The active programs might be encapsulated inside a user VM, uploaded without a VM (so-called *standalone user APs*), or provided by the DiProNN (*standalone built-in APs*). Similarly, the VMs, which serve as execution environments for the standalone APs, could be uploaded by the users, or provided by the DiProNN as well.

Every session has to be established before the data transmission and processing might begin—the session's establishment itself consists of a delivery of a new session request, a verification of resource requirements' satisfaction, resource allocations, uploading of all the necessary APs/VMs, and setting all the communication channels (see the Section 7.3). Such an established session can be further used for the processing of passing user data. At the end of the processing, the session has to be terminated, all the allocated resources released, and all the uploaded APs/VMs destroyed (if not requested otherwise).

6.2 DiProNN Programs

As depicted in the beginning of this chapter, the DiProNN Programs define the processing applications (the DiProNN Sessions). More precisely, they define at least all the APs and VMs required for the particular application (including the resource requirements' specification and other necessary information), the data and control flows among all the processing APs, and another necessary information related to the session. Such a definition can be based on a complex and well-known jobs' description language (e.g., the *Job Submission Description Language (JSDL)* [17]), or based on a simple pseudo language we use throughout this thesis, whose general structure is depicted in the Figure 6.1.

DiProNN Session Graphs—the graphical representations of the DiProNN Programs—then serve mainly for an illustration, simpler design, and better understanding of the overall sessions' functionality and the data flows defined among their APs (see the Session Graphs' symbols depicted in the Figure 6.2). Such a graphical representation properly accommodated with desired Sessions'/APs'/VMs' options and created using an appropriate editor might also serve for DiProNN Programs' automatic generations.

[1] Remember, that the DiProNN operates on a private network segment—once a VM is migrated to a different node, it must be referenced by the remote node's public IP (further details about the migrations are provided in the Sections 7.3 and 7.4.2).

[2] Note, that such a forwarding is attainable since the Service Domain can "see" and control all the packets, that the VMs, which it controls, send (even the ones not destined to it).

```
# Session's parameters section
# (owner, notifications, overall resource requirements, ...)

# APs and VMs used for the processing follow
# ------------------------------------------

# standalone APs
{ AP name="first_AP" ref="first_AP_reference";
    # AP parameters
    inputs = ...;
    outputs = ...;
    # EE's requirements
    # resources' requirements
    # ...
}

# ... other standalone APs


# VMs including APs
{ VM  name="my_VM1" ref="my_VM1_image";
    # VM parameters
    { AP name="thisVM_AP" ref="thisVM_AP_reference";
        # AP parameters
        inputs = ...;
        outputs = ...,
        # ...
    }
    # ... other APs
}

# ... other VMs
```

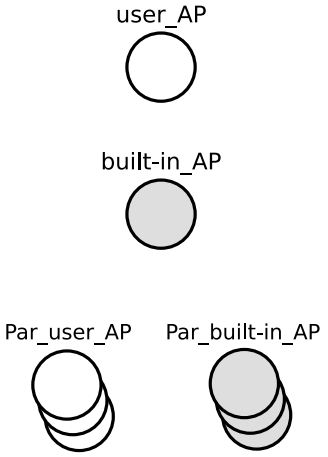
Figure 6.1: The general structure of the DiProNN Programs written in the pseudo language.

6.2.1 (Standalone) APs' definition

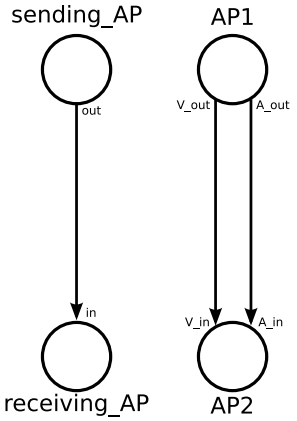
As already depicted, there are two kinds of the standalone APs in the DiProNN—the ones uploaded by users and the built-in ones provided by the node. Besides these standalone APs, as we show later, another APs might be also directly included inside the users' VMs.

The intended pseudo language characterizes all the APs by two parameters—the `name` parameter and the `ref` parameter. Whilst the `name` parameter provides a unique identification name of the particular AP (unique in the context of the particular session) and serves for referencing the APs in the communication channels' definition (see the Sections 6.2.3 and 6.2.4), the `ref` parameter identifies the relevant (compressed) file, which contains the uploaded standalone user's AP (e.g., *ref="filename.tgz"*), the particular built-in AP distinguished by a specific keyword (e.g., *ref="DiProNNservice.AP_ident"*), or the AP included inside an uploaded VM (e.g., *ref="included_AP_ident"*)[3].

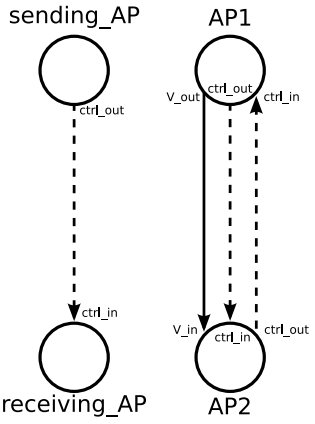
[3]Note, that the AP must be further identified by the VM's APM service, which has to have all the necessary information related to the included APs set (e.g., real locations, startup commands, etc.).



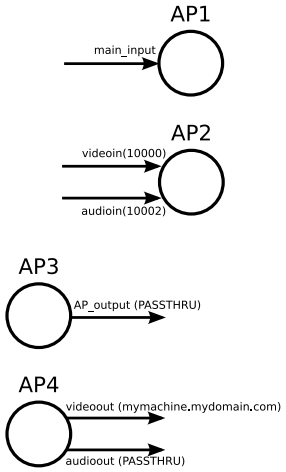
(a) User AP, built-in AP, and parallelizable AP symbols.



(b) Data channels' symbols.



(c) Control channels' symbol.



(d) DiProNN input and output symbols.

Figure 6.2: DiProNN Session Graph symbols.

For the standalone APs, the other important parameters are the required EE's identification/description and the resource requirements specification (the requested resources are, as we show later, provided to the VM the particular AP runs in). Moreover, all the APs could be supplemented with additional parameters, which the particular implementation allows/requires to define (e.g., startup command, command-line arguments, processing adjustments, etc.).

6.2.1.1 Parallel Processing

As we have already mentioned, the proposed node is able to process the active programs in parallel. Even though the detailed description of the parallel processing itself is provided in the Section 7.4.1, the definition of such a parallelizable AP in DiProNN Programs is described here. Thus, if a special AP's attribute

```
parallelizable[(instances_count)]
```

is defined, the DiProNN performs the AP's processing in parallel. The number of running parallel instances can be either fixed (the requested `instances_count` is present and correctly specified), or variable (controlled by the DiProNN Session Operator module according to the actual AP's load and resources available).

However, once a user requires processing of an active program in parallel, he or she has to specify, how to distribute the incoming data stream over such parallel instances. Thus, before every AP having the parallelizable attribute set, there must be an AP (built-in or user-loaded) defined, which performs the data distribution over the parallel instances (further denoted as the *distribution active program*).

6.2.2 VMs' definition

Similarly to the APs, the node assumes two kinds of the VMs as well—the user ones and the built-in ones. Whilst the built-in ones need not to be explicitly defined in the DiProNN Programs (they are just specified in the identification/description of the EEs required by the standalone APs), the user ones have to be explicitly defined.

Against to the APs, the user VMs are characterized by a single parameter only—the `ref` parameter, which identifies the relevant VM's image. And similarly to the APs, the users might specify the overall resources requested by the particular VM and all the included APs as well as additional parameters, which the particular implementation allows/requires to define.

As already depicted, the users' VMs, that are intended to be uploaded into the node, might directly include users' APs. These APs have to be defined within the particular VM's section (see the Figure 6.1) in the same way as the standalone APs are (obviously, without resources' and EEs' requirements). Note, that the users are also able to define the built-in standalone APs within the VM's section—once a particular built-in AP is requested to be run inside a user VM/EE, it is uploaded into the VM during sessions' establishment (details about sessions' establishment process are provided in the following chapter).

6.2.3 Data Interfaces' and Channels' Definition

We have already mentioned, that all the active programs are referenced by their unique identification names. Similarly, all the input/output data/control interfaces, which describe the communication channels among the active programs required for the session's processing, are referred by their identification names as well. These symbolic names are, as soon as the APs are started, associated with real network port numbers, that the APs communicate throught.

6.2.3.1 Inputs

Within the AP's parameters section, there must be a single parameter (named `inputs`) specified, which defines the AP's data input interfaces, and which has to be of the following form:

```
inputs = input1_name[(DIPRONN_INPUT[(port1)])],
         input2_name[(DIPRONN_INPUT[(port2)])],
         ...
```

where

- the `inputX_name` specifies the symbolic name of the particular input data interface, and
- the `portX` is an optional parameter specifying the requested port number of the particular DiProNN's data input (the external one).

Whilst the input of the simpler form (`input_name`) specifies the AP's input interface, which is further referenced by another AP (its output port), the input of the form `input_name(DIPRONN_INPUT(port))` means, that the input, named `input_name`, is the node's input (the input, which the user sends data to). Such an input might define a particular `port`, which the DiProNN should listen on[4], or might let the DiProNN to choose an arbitrary port, which becomes negotiated during the session's establishment.

Examples:

- `inputs = audio_in1, video_input;`
- `inputs = in1(DIPRONN_INPUT);`
- `inputs = in3(DIPRONN_INPUT(10000));`

6.2.3.2 Outputs

Within the AP's parameters section, there must be a single parameter (named `outputs`) specified, which defines APs' data output interfaces and the data communication channels themselves. The parameter has to have the following structure:

```
outputs =
   output1_name(AP1name.in1 | DIPRONN_OUTPUT(rcvr1:port1 | PASSTHRU)),
   output2_name(AP1name.in2 | DIPRONN_OUTPUT(rcvr2:port2 | PASSTHRU)),
   ...
```

where

- the `outputX_name` specifies the name of the output interface, and

[4] As already depicted, the DiProNN inputs are opened on the node's Distribution Unit, where the incoming data are forwarded to the particular AP. In the case of distributed processing described in the Section 7.3 and the Chapter 8, the particular implementation might decide, whether these inputs are opened on the entrance node or on the particular remote node.

- the `APXname.inX` specifies, that the data leaving the particular AP throught the `outputX_name` output interface have to be forwarded to the input port (named `inX`) of the subsequent AP (named `APXname`).

The output interface, which is specified in the form

```
output_name(DIPRONN_OUTPUT(receiver:port | PASSTHRU))
```

then means, that the output interface, named `output_name`, is the DiProNN's data output. The data receiver is specified either by an IP address/domain name and a port number, or (in the case of the `PASSTHRU` keyword) is defined during the data processing by the particular AP itself. In the latter case, the DiProNN lets the outgoing data packets as are—it does not perform any forwarding, so that the data pass throught the Aggregation Unit and continue to the receiver, which is set by the AP (further details about the DiProNN's data flows are provided in the following chapter).

Examples:

- `outputs = audio_out(previous_AP.audio_in1), video_out(specialized_AP.video_input);`
- `outputs = output(DIPRONN_OUTPUT(comp.mydomain.com:9738));`
- `outputs = out(DIPRONN_OUTPUT(PASSTHRU));`

6.2.4 Control Interfaces and Channels' Definition

As the Section 4.6 describes, there are two interconnections assumed in the DiProNN—the data one and the control one. Whilst the data interconnection serves for data transmissions among the APs, the control one serves for internal control communications among them—to indicate some events, that have occurred in the data stream, to share information about actual state, or just to ask another AP for something. In such cases, the users may define the control communication channels among the APs similarly as they do it for the data ones—using the keywords `control_inputs` and `control_outputs`. Nevertheless, the special forms `DIPRONN_INPUT(...)` and `DIPRONN_OUTPUT(...)` are not allowed for the control messages.

Thus, as soon as a particular communication channel is defined as the control one, all the messages passing through it are sent via the internal (low-latency) control interconnect[5].

6.2.4.1 Control communication among the parallelizable APs

Regarding the control communication in cases, where at least one communication partner is a parallelizable AP, there are three possible scenarios:

1. *parallelizable AP is a sender of control messages* – when a parallelizable AP has to be able to send control messages to another AP, the communication channel between both the APs is defined in the same way as the ones, that define the control communication between non-parallelizable APs.

[5] If there is any. When there is just one interconnection used for data and control communication simultaneously, the control messages are sent in the same way as the data ones are.

2. *parallelizable AP is a receiver of control messages* – if an AP has to be able to send control messages to a parallelizable AP, the communication channel is also defined in the same way as the one defining the control communication between non-parallelizable APs. However, in this case, the control messages are broadcasted over all the parallel instances of the destined AP.

3. *control communication among the parallel instances* – when the parallel instances want to communicate with each other, the combination of the previous two points takes place: the particular parallelizable AP has to define a control output, that is connected to a control input of the same AP (see the definition of the `transcode` AP in the example 8.2). In this case, all the control messages coming from the specified output control interface are broadcasted over the appropriate input control interfaces of all the other parallel instances of the given AP.

6.3 Example: Video Streams' Composition in DiProNN

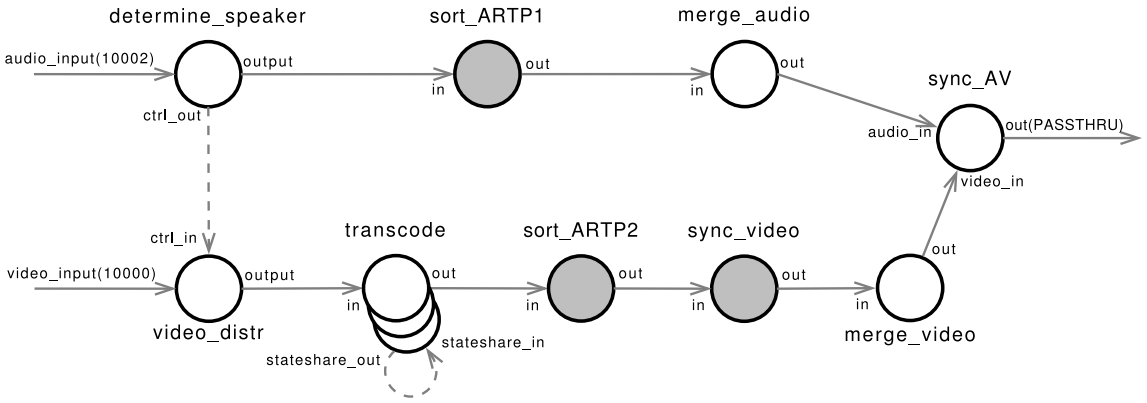
In this section, we sketch a possible implementation of a video streams' transcoding and composition application described in the Section 10.2. The application, which aims to merge/compose incoming multimedia streams into a single output stream with current speaker somehow highlighted (greater picture and/or lighter-colored), is sketched using the depicted pseudo language and illustrated on the DiProNN Session Graph—see the Figure 8.2.

Let us assume, that all the streams are transferred using the ARTP transport protocol, which is able to distinguish among the input streams on the application-level (e.g., a stream identification is passed as an ARTP option). Moreover, both the video and audio data are transferred in an independent data blocks, as the ARTP assumes.

At first, there is a current speaker determined[6] from the incoming audio stream (this is performed by the `determine_speaker` active program). Once determined, the identification of the current speaker is sent to the `video_distr` AP using the control interconnection, where a relevant option indicating the current speaker is added to his/her video stream. This option is read by the `transcode` AP[7], which highlights the current speaker's video before/after the transcoding and forwards the stream to the `sort_ARTP2` AP performing the independent data blocks' proper ordering. Once properly ordered, all the transcoded video streams should be synchronized and merged into a single video stream. Finally, as soon as a single video stream is created, it should be synchronized with the merged audio stream and sent to the receiver(s).

[6]The real method of a speaker's recognition is not important for this example—several possible methods could be found in [228].

[7]Note, that the `transcode` AP might be processed in parallel and each parallel instance may communicate with the others, as indicated in the DiProNN Program.




```
# session's parameters (owner, notifications,
# overall resource requirements, ...)

{AP name="determine_speaker" ref="recognize_speaker1.tgz";
  # AP parameters
  inputs = audio_input(DIPRONN_INPUT(10002));
           # requested DiProNN input port is 10002
  outputs = output(sort_ARTP1.in);
  control_outputs = ctrl_out(video_distr.ctrl_in);
}
{VM ref="my_VM1.img";
  # VM parameters
  {AP name="transcode" ref="transcode_video";
    inputs = in;
    control_inputs = stateshare_in;
                     # stateshare_in is the input for
                     # communication among parallel instances
    outputs = out(sort_ARTP2.in);
    control_outputs = stateshare_out(transcode.stateshare_in);
    parallelizable;      # parallelizable AP
  } # ... other APs
}
{VM ref="my_VM2.img";
  {AP name="sync_AV" ref="syncerAV";
    inputs = audio_in, video_in;
    precision = 0.001;  # 1ms
    outputs = out(DIPRONN_OUTPUT(PASSTHRU));
              # the real receiver is defined inside the ARTP packets
  } # ... other APs
} # ... other APs/VMs
```

Figure 6.3: The DiProNN Program's fragment and the DiProNN Session Graph of a video streams' composition application.

Chapter 7

DiProNN Operational Overview

The previous chapters have presented the DiProNN's architecture, the data and control protocols, that are used for data transmissions and (internal) communication, and the programming model, which is used for the DiProNN programming. So far, there have been presented just a few pieces of information regarding its operational functionality itself. Nevertheless, this chapter redresses it—in the following sections, we present and define the DiProNN's operation in detail, starting with its initialization through a new session establishment, data flows and session's processing, and ending with an established session's termination.

7.1 Initialization

The DiProNN initialization means the process, which starts with starting all the DiProNN units', and which ends in the moment, when the node is ready for an establishment of new sessions. It is assumed, that once started, the units do have their network stacks on both the data and control interconnections initialized, either via the DHCP or similar network service.

During the initialization, all the DiProNN units have to start their necessary service modules at first. Once started, all the units (except the Control one) have to register themselves at the Control Unit (the Section 7.1.1), which subsequently discovers all the resources available on them (the Section 7.1.2). Finally, the Control Unit asks selected Processing Units for obtaining and starting the virtual machines, that are intended to be started during the initialization. Once the Control Unit receives an information, that the VMs have been started, it sets all the appropriate forwarding rules at the Distribution, Aggregation and relevant Processing Units, so that the VMs become accessible from outside (if necessary) and the relevant data can pass properly. These forwarding rules are set in a similar way as the sessions' forwarding rules described in the Section 7.3.

Note, that the units' registration approach, which is described in detail later in this section, enables the DiProNN administrators to add and register new Processing Units during the node runtime. The administrators are thus able to response to requirements of additional resources or to response to sudden failures of the units, without any needs to restart the whole node and completely reinitialize it (which would have resulted in terminating all the already established sessions).

For simplification purposes, the graphical representation of the whole initialization process is depicted in the diagram in the Figure 7.1. The particular parts of this process are further discussed in detail in the following sections.

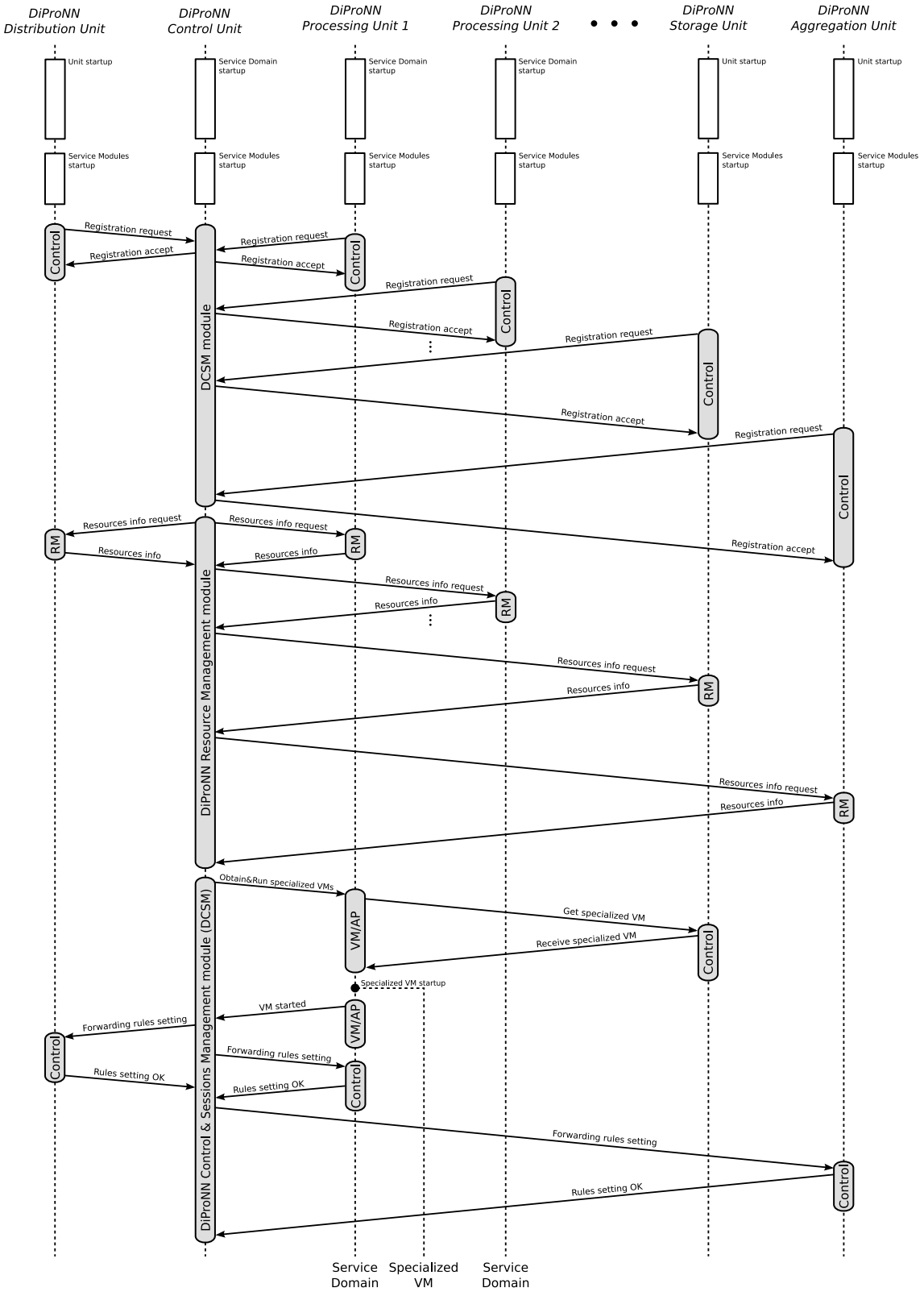


Figure 7.1: DiProNN initialization diagram.

7.1.1 Units Registration Process

At the beginning of the initialization process, all the units have to boot up and start all the necessary service modules running inside of them (the VM/AP Management module, the Control module, the Resource Management module, etc.). As soon as started, the units perform the registration process—they register themselves at the Control Unit, which thus becomes aware of all DiProNN's physical components (units), that can be used for processing the sessions. The registration, which is temporary and must be refreshed every specified time period, is performed by periodical registration requests sent from the units (from their Control modules) to the DiProNN Control & Sessions Management (DCSM) module[1].

As soon as the Control Unit accepts the request, the positive reply is sent to the requesting unit's Control module, and the process of units' available resources discovery follows.

The liveness of all the already registered units should be periodically checked by the Control Unit (by contacting units' Control modules). If a unit does not respond to liveness requests, it is said to be functionless and thus removed from an internal database. Simultaneously, all the users of influenced sessions (the sessions, that have an AP running on the not responding unit, or all the users in the case of Distribution or Aggregation Units' failure) are informed. The influenced sessions are subsequently terminated (see the Section 7.5).

7.1.2 Units' Resources Discovery Process

After the successful registration, all the Resource Management modules, which run on all the registered units, are contacted by the DiProNN Resource Management module, which asks them for an information about their available hardware resources. This information is periodically obtained by the module and, as already mentioned in the Section 4.3, serves mainly for decisions about placing new APs/VMs, VMs' migrations, and resources' usage monitoring.

For the resources discovery and information exchange among the units, the use of a standardized monitoring software tool is highly desirable. The examples of such tools are the *Ganglia* [232]—a scalable distributed monitoring tool for high-performance computing systems such as clusters and Grids—and the *GLUE schema* [16], which is an information model for Grid entities described using a natural language and enriched with a graphical representation using UML Class Diagrams [33]. Such a resources discovery and monitoring software tool has to collect, and to the DiProNN Resource Management module provide an information (status and utilization) at least about all the unit's CPUs, network I/O, disk I/O, and memory subsystem. The acquired information is then managed by the module, and if required, provided to other ones.

[1]Note, that the registration process assumes the Control Unit being already started and fully functional. Nevertheless, if the Control Unit is not ready for responding to unit's registration requests in the particular time, the requesting units do not receive a notification of accepting the registration in a specified timeout and thus keep to resend the registration requests until any reply from the Control Unit is received.

7.2 Users' Requests

As soon as the DiProNN is initialized, it is able to respond to users' node-related messages/requests, which have to be delivered using the DiCP protocol (described in the Section 5.2.2) to the DiProNN Control Port established on the node's Distribution Unit. Before the node processes the requests, it may require the users to authenticate depending on the access policy used. The authentication itself is performed by the DiProNN Access Management module—the users have to authenticate themselves using, e.g., shared secret knowledge, registered username and password, public key infrastructure (PKI) authentication using X.509 certificates, etc.

Once the users are authenticated, the node processes their requests depending on their authorization. The authorization policy—the definition, which users (and/or user groups) are able to request which requests/functionalities—must be defined by the DiProNN administrator(s). The authorization process itself is then performed by the DiProNN Access Management module as well.

In general, the requests, which are accepted on the DiProNN Control Port and which are further delivered to and processed by the DiProNN Control & Sessions Management module, can be divided into two groups—the *informational requests* and the *controlling requests*. For example, using the informational requests, the users might obtain an information about the actual state of the node, built-in functionality provided, or previously established DiProNN Sessions (logged history). The controlling requests serve for authorized DiProNN administrators to remotely control the behavior of the node (e.g., forcing termination of established sessions, security and access policy setting, controlling of DiProNN's HW and SW resources, etc.), and for authorized users to ask for a new DiProNN Session establishment, as described in the following section. Further details about the external DiProNN's management are provided in the Section 5.2.2.

7.3 Session Establishment

As already mentioned in the previous section, once the users pass the authentication process, they are able to ask the node for a new session establishment. These requests, which have to contain the session's description (DiProNN Program), are delivered to the DiProNN Control & Sessions Management module, which accepts or refuses them.

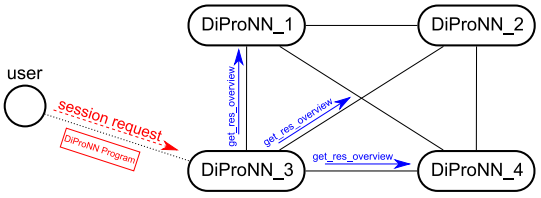
As soon as such a request is delivered to the module, it decides in cooperation with the DiProNN Resource Management module, whether the resource requirements could be satisfied or not. The decision depends on the actual DiProNN usage (resources available vs. resources required) and also comprises of the APs/VMs mapping process (see the Section 7.3.1)—the decision, which VMs the standalone APs will run in and which Processing Units each VM will run on.

If the session request could be satisfied by the node itself, all the requested resources are reserved (if any). Subsequently, the appropriate Session Control Port is created, and the DiProNN Session Operator module, which is used for managing the session being established, is started. Finally, the user is informed about the session's acceptance (including the information about the Session Control Port opened) and the session establishment process takes place (see the Section 7.3.2).

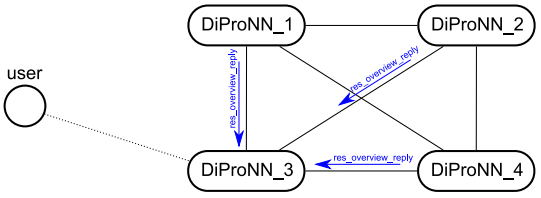
If the request could not be satisfied by the particular node itself, its DiProNN Control & Sessions Management module tries to ask the other known[2] DiProNN nodes in

[2]Known, e.g., via an external service managing all the DiProNNs in the network, and providing a relevant information about them.

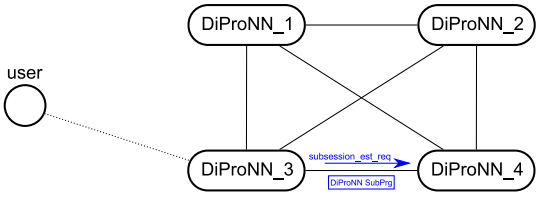
the network for participating on the session's processing. At first, it asks them for an overview of their available resources—see the Figure 7.2(a). As soon as the overview is received (Figure 7.2(b)), the module decides, whether there is a "better" DiProNN node (e.g., located closer to session's data receivers, less loaded, more powerful, etc.), which is more suitable for the session's processing, or not. In the positive case, the DiProNN user is asked for sending the session request to the chosen node, while in the negative case, the module tries to divide the original session into several *subsessions*. Once the session is divided, the module asks each of the selected nodes to establish the relevant subsession (Figure 7.2(c))—in fact, it asks for a new session establishment in a common way, but just with a relevant part of the original DiProNN Program. This process is performed until all the nodes, which have been asked to establish the subsessions, confirm its establishment, or there is a possibility to find another suitable session's distribution.



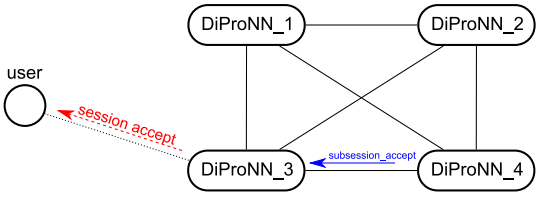
(a) Incoming new session establishment request and asking another DiProNN nodes for an overview of available resources.



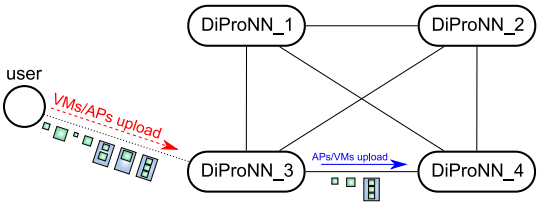
(b) Answers to the resources overview requests.



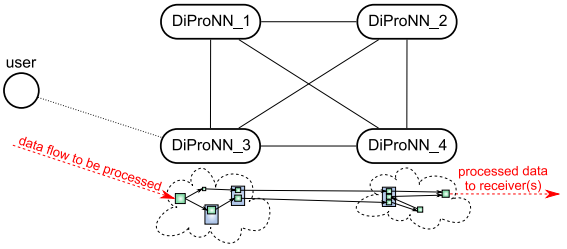
(c) Request to participate on the session's processing—the subsession establishment request.



(d) The subsession establishment accept.



(e) The process of VMs/APs uploading.



(f) Data flow and distributed processing.

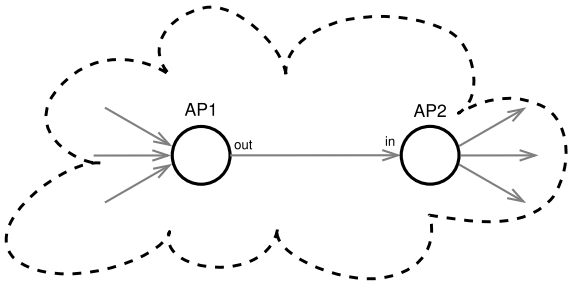
Figure 7.2: The process of requesting another DiProNN nodes to participate on a new DiProNN Session processing.

If all the nodes, which have been asked to participate on the session's processing, accept the subsessions[3], an acceptation to the session establishment request is sent to the user (Figure 7.2(d)) and the uploading of the VMs/APs (Figure 7.2(e)) followed by

[3]Note, that each of the nodes, which has accepted the subsession, had to perform the APs/VMs mapping process as well as resources reservations before accepting it.

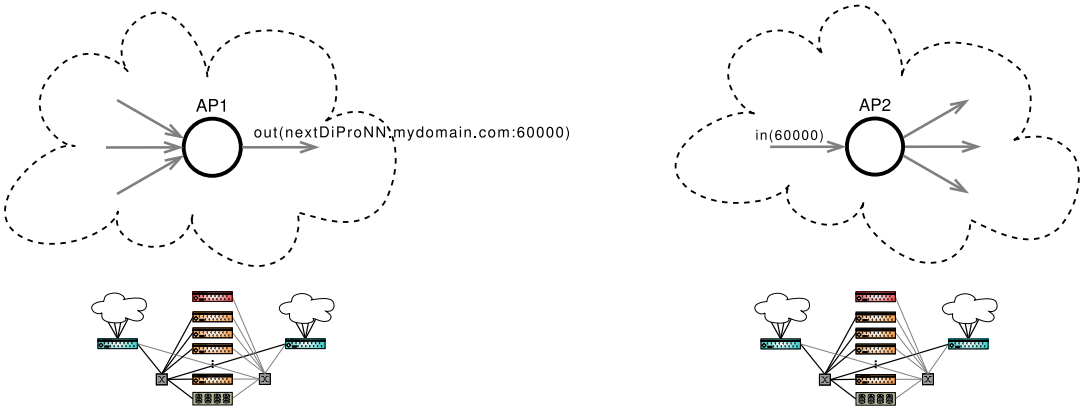
the data transmission and processing (Figure 7.2(f)) might begin. Otherwise, the session establishment is rejected.

Finally, let us point out, that once the session is divided into several subsessions, the data communication channels[4] among the divided APs—the last AP(s) from one subsession and the first AP(s) from the following one—have to be transformed. This is necessary since the Processing Units operate on a private network segment, and thus are not directly accessible from the other nodes—otherwise, the Control module of the Processing Unit, where the sending AP(s) from a particular subsession runs, would have not been able to deliver the data to the following AP(s) running in the following subsession(s).



AP1: `outputs = out(AP2.in);`
AP2: `inputs = in;`

(a) Before the transformation (single subsession).



AP1: `outputs = out(DIPRONN_OUTPUT(nextDiProNN.mydomain.com:60000));`
AP2: `inputs = in(DIPRONN_INPUT(60000);`

(b) After the transformation (two subsessions).

Figure 7.3: The transformation of data channels divided into two subsessions.

The transformation of the data channels, that have been divided into two subsessions, simply changes the outputs of the sending AP(s) and inputs from the receiving AP(s) in the way depicted in the Figure 7.3—the receiving APs' inputs are changed into

[4] As the Chapter 8 shows, the division of the control communication channels need not to be considered, since the sessions cannot be divided between the APs communicating via the control interconnection.

`DiProNN_INPUT`s, which results in opening this ports on the Distribution Unit of the node processing them (making them accessible from outside the node), and the sending APs' outputs are changed into `DiProNN_OUTPUT`s with the following DiProNN node and the appropriate port number set as the receiver. If a subsession contains an AP(s) with data/control DiProNN inputs, the particular DiProNN implementation may choose between two approaches:

- *providing users with an information about the nodes processing the subsession(s)* – in this case, the ports of the DiProNN inputs are opened just on the DiProNN node(s) processing the relevant subsession(s). The users are then provided with an access information of every opened port—the hostname or the IP address of the DiProNN node processing the subsession and the port number opened on its Distribution Unit, through which the relevant input port is accessible.
- *hiding the distributed processing (and the other DiProNN nodes)* – if the distributed processing has to be hidden because of some reason, besides opening the input ports on the relevant nodes processing the subsessions[5] the same number of ports must be opened on the DiProNN node, which the user is communicating with (further denoted as the *entrance node*). In this case, the users are provided just with an information of the port number(s), that have been opened on the node establishing the session (the entrance node), whose Distribution Unit forwards all the incoming data to the particular node(s) really processing the relevant AP(s).

7.3.1 APs/VMs Mapping Process

Once the nodes, that are able to provide requested resources to satisfy session's requirements, are chosen, each of them has to decide, which VMs will all the standalone active programs run in, and which Processing Units will all the VMs run on. This process is, as depicted before, called the *APs/VMs Mapping Process*.

The APs/VMs mapping process has two goals. First, it has to choose proper execution environments for all the standalone active programs (both user and built-in ones), that participate on the session's processing—depending on their requirements (e.g., OS type, libraries available, etc.), it chooses the appropriate built-in EEs for them. The particular implementation might define, whether several APs might be planned into a single EE. However, not to degrade the security benefits provided by the virtualization, in such a case the node should not plan APs belonging to different sessions into a single EE.

The latter goal is to distribute the resulting VMs over the Processing Units of the particular node(s) in a way, that their resources requirements are met. If necessary, already running VMs might be also migrated (as described in the Section 7.4.2), so that the requested resources are freed.

If a suitable mapping is found, all the requested resources are allocated (so that another session cannot use them) and the session request is said to be satisfiable. Then, the positive response is sent to the user, who is subsequently invited to upload session's APs/VMs—the session establishment process, which is described in the following section, thus takes place.

[5]These ports have to be opened, since the APs must be accessible from outside. However, the data, which has not been sent from the participating DiProNN node, should be discarded.

7.3.2 Session Establishment Process

As soon as the user receives a notification about accepting the establishment request, he or she informs the DiProNN Session Operator module via the Session Control Port about starting the uploading of the virtual machines and/or active programs related to the session. The APs/VMs could be uploaded to every node, which participates on the session, individually, or just to the entrance node (see the Figure 7.2(e)), which further uploads the relevant APs/VMs to the subsequent node(s) in the same way as the user does (it behaves like a common user to the subsequent node(s)).

The uploading itself is performed using a specialized service (e.g., the FTP, SCP/SFTP, or FTPS) to a location dedicated on the Storage Unit, which has been specified during the (sub)session establishment. Once the uploading finishes, the DiProNN Session Operator module of the particular node informs the selected Processing Units' VM/AP Management modules about the (sub)session's APs/VMs, which will the particular unit run (based on the plan resulting from the APs/VMs mapping process). Each VM/AP Management module then obtains the relevant APs'/VMs' files/images (including the built-in ones, if requested) from the Storage Unit; the obtained VMs are subsequently started.

As soon as the VMs are started, a communication between the VM/AP Management module and the APM service of every session's VM follows. The goal of the communication is to upload all the necessary standalone APs into the particular VM[6] and according to the particular DiProNN Program to instruct the particular APM service, which APs and how have to be started. Once provided, the relevant APs are started.

Communication Interfaces' Association

During the APs' startup, every AP has to associate its input/output data/control interfaces with real network port numbers, which the particular communication interface will belong to. The particular implementation might decide, whether such an association will be performed in a static or a dynamic way. The static association means, that the particular APs' interfaces are statically assigned with appropriate (and for every startup the same) network port numbers. As opposed to it, the dynamic association chooses the appropriate port number for the particular interface dynamically depending on current conditions during the establishment (it associates the interfaces with currently free network ports). As obvious, the static associations are simpler and do not require the APs to perform any communication in order to associate the interfaces, however, the dynamic ones are more flexible and are desirable especially for implementations, that allow more active programs to run in a single VM (this prevents port collisions, which might appear, when more APs require a particular network port).

While the static associations could be defined in the APs' configuration files, the dynamic associations are performed using the Port Associator (PA) service—for each input/output data/control interface, which the AP wants to communicate with, the AP sends the interface's name to the PA service and asks, which network port number the relevant interface should belong to. The PA service performs the association and replies with the network port number, that the relevant interface has to connect to. Simultaneously, the information about all the couples *(communication interface name, real port number)* is stored, and as soon as all the APs are started and have their interfaces associated, it is sent to the Control module of the particular Processing Unit.

[6]Note, that the user VMs might contain the user APs directly included. In this case, no uploading is necessary.

Subsequently, as soon as all the couples are registered, the Control module uses this information for setting the communication channels as described in the following paragraphs.

Communication Channels' Creation
As soon as all the APs, which are required for the session being established, are started and as soon as all the communication interfaces are associated with real network port numbers, the creation of the communication channels follows. The DiProNN Session Operator module obtains the information about all the couples of the associated communication interfaces from the Control modules of relevant Processing Units and depending on the DiProNN Program creates a set of forwarding rules that have to be applied to ensure proper data flows among the APs. Once the set is created, its relevant part is forwarded to the Control modules of the particular Processing Units as well as to the Control modules of the Distribution and Aggregation Units, where appropriate forwarding rules are set as well.

The forwarding rules reflect the communication channels, that are specified in the DiProNN Program. For example, when an active program named A wants (through its output interface associated with the port number a) to send data to an active program B (to its input interface associated with the port number b), the forwarding rule, that has to be set in the Service Domain of the relevant Processing Unit, forwards all the packets coming from the relevant VM (determined by its IP), which the A active program runs in, and having their source port number set to a, to the IP of the VM running B active program and to the b network port (to the successive AP listening there). The forwarding process itself depends on the particular DiProNN implementation, however, the use of kernel *Netfilter*[7] (`iptables` [15]) is usually highly advisable (see details in the Chapter 11).

As soon as the communication channels are created, the DiProNN Session Operator module(s) in cooperation with the Resource Management modules of the relevant units reserves the requested resources[8]. After that, the session becomes established and ready for data transmissions and processing—the user is informed about successful session's establishment and invited to send data. If an error occurs during the establishment process, the user should be also informed together with an information about error details.

The graphical representation of the whole session's establishment process is depicted in the diagram in the Figure 7.4.

7.4 Data Flow and Processing

Once a new DiProNN Session is established, the data flow through the node could be briefly described in the following way: when a user packet arrives to the Distribution Unit, it is forwarded to the Processing Unit, which processes the session's first AP (more precisely, to an network interface of the relevant virtual machine and to the registered network port, which the particular AP listens on). The forwarding is performed according to the forwarding rules previously set.

[7] `http://www.netfilter.org/`

[8] If some network parameters have to be guaranteed between two nodes in the case of distributed processing, these cannot be ensured by the DiProNN itself—these must be provided in cooperation with the relevant services of the public networking infrastructure.

As soon as the packet is processed, it leaves the AP throught a specific output port. The forwarding rules apply again and the packet is forwarded to the next active program(s) for further processing. This procedure repeats until the packet traverses all the APs running on the particular node, which have to process it. After the whole processing, the packet is forwarded throught the Aggregation Unit to the receiver. This forwarding depends on the situation, that occurs:

- *Data receiver is specified in the DiProNN Program* – if the data receiver is specified directly in the DiProNN Program (see the Section 6.2.3.2), the Service Domain of the Processing Unit, which runs the last processing AP, forwards the packet leaving the AP through a particular output port directly to the data receiver.
- *Data receiver is specified by the last processing AP* – if the data receiver is specified by the last processing AP (the `PASSTHRU` keyword is used in the output channel's definition, as described in the Section 6.2.3.2), the Service Domain lets the packets leaving the last AP as they are—the packets leave the Processing Unit and through the Aggregation Unit continue to the receiver set by the AP.

Note, that this data flow approach conforms to the distributed processing as well. As soon as the session is distributed among several nodes, a set of subsessions is created according to the transformation described in the Section 7.3. Thus, each subsession could be considered as a common DiProNN Session, which has some input channels, some output channels, and output channels' receivers set (pointing to the session's receiver(s) in the case of last subsession or to the subsequent DiProNN(s) otherwise).

Nevertheless, such a data flow approach does not enable the active programs to be able to determine the IP address of the real data sender/receiver (if required)[9]—each packet is destined by the particular Service Domain to a relevant VM's address and given AP's port, and thus has its source/destination IP address/port set. To make their active programs being aware of the real data sender/receiver, the users have to use an extended functionality of the data transport protocol used—e.g., protocol's data options or user-specified data format.

7.4.1 Parallel Processing

As depicted in the Section 6.2.1, the DiProNN is able to run active programs in parallel. It means, that the virtual machine, which runs the parallelizable AP, is mirrored over several Processing Units of a single node[10], where the passing data are simultaneously processed. The resource requirements, which have been specified for the particular parallelizable AP in the DiProNN Program, are then applied for every parallel instance in the amount requested (except the requested network bandwidth, which is portioned appropriately).

Nevertheless, this approach has two main prerequisites to enable an active programs to run in parallel. First, the data being processed have to be separable into independent data blocks (for example, the ARTP datagrams independent on each other), so that

[9] If the Distribution Unit simply forwards all the incoming packets, the first AP could be able to determine the data sender. However, once the Distribution Unit has to provide some precomputations on the incoming packets (e.g, extractions of packets encapsulated using the IPSec [148] protocol), the information about the real data sender could be lost (the unit becomes the data sender from the AP's point of view).

[10] All the parallel instances of a single AP have to run on the same node, since they should be allowed to communicate with each other through the (low-latency) control interconnection as fast as possible. Otherwise, the communication latency would have been increased by the external interconnect (e.g., the Internet).

they can be spread over the parallel instances, where they are processed independently on each other. Second, before each parallelizable AP, there must be a *distribution active program* specified—the active program, that collects all the independent data blocks and distributes them over the parallel instances.

If both conditions are met, users may specify the parallelizable APs using the relevant option in the DiProNN Programs (`parallelizable[(instances_count)]`—see the Section 6.2.1). If the instances' count is specified, the parallelizable AP is mirrored by the DiProNN Session Operator module over specified number of Processing Units during the session's establishment. If the instances' count is missing, the DiProNN tries to adapt the number of instances to current processing conditions (whether the running instances can cope with the amount of data, that have to be processed)—during the session's establishment, the AP is mirrored over an initial number of the units.

To properly distribute the incoming data, the distribution active program has to be aware of the number of AP's parallel instances, that might be used in that particular time. Thus, during its startup (more precisely, during the process of communication interfaces' association), the relevant PA service reserves the appropriate number of real network port numbers for every distribution AP's output port (equal to the `instances_count`, if specified, or being at least[11] the same as the current count of the Processing Units is), and the distribution active program shares a constant/variable with the APM service, that indicates the current number of parallel instances, over which the data might be distributed (being set statically to the `instances_count` or having an initial value and varying in time). Subsequently, all the forwarding rules are set—each real network port must be addressed to a single and unique parallel instance.

However, if the node has to adapt the number of parallel instances to current processing conditions, a detection of instances' saturation must be performed. Such a detection process depends on the particular DiProNN implementation—for example, depending on the amount of data being distributed, the distribution active program may decide to ask the DiProNN Session Operator module for a new parallel instance (the request is delivered throught the APM service and the VM/AP Management module) or the APM service of each parallel instance may monitor the amount of packets, that have been dropped by the particular OS's kernel (e.g., via the `tcpdump` tool). If there are some packets dropped for a specified amount of time, it can be supposed, that the current number of instances is unable to cope with the amount of data required to be processed—the APM service may thus ask the DiProNN Session Operator module to increase the number of parallel instances as well. Nevertheless, the final decision is up to the DiProNN Session Operator module, where all the requests are forwarded.

7.4.2 VMs' Migrations for Efficient Resources Utilization

The VMs' migration process occurs, when the DiProNN Control & Sessions Management module decides to redistribute the running VMs from any reason. Such migrations could be generally performed either inside a single DiProNN node (let us call them *intra-node migrations*) or among different nodes as well (*inter-node migrations*). For example, the intra-node migrations could be performed to utilize node resources in a more efficient way (the goal is to apply a better distribution that enables the node to accept sessions, that would have been rejected because of unavailable resources before the migration), whereas the inter-node migrations could be also performed because of better resources' utilization, but also because of service repairs required to be performed on a particular

[11] The PA service may reserve more real port numbers to allow future adding of the Processing Units'.

node, or because of more effective usage of the network (e.g., the session distributes incoming data and most receivers are located nearer a different node).

The migration itself is provided by the relevant DiProNN Session Operator module (or modules in the case of inter-node migrations), which the migrated VM belongs to. However, since the inter-node migration should be also enabled to the nodes, which do not participate on the session's processing, the DiProNN has to cope with the non-availability of a dedicated DiProNN Session Operator module running there (the one, that belongs to the particular session). Since this module is necessary during the migration process, at first, the remote node has to be asked for an establishment of a new empty DiProNN Session (a session without any APs/VMs). Once this empty session becomes established, a dedicated DiProNN Session Operator module is started on the remote node, and the inter-node migration process might be performed.

As depicted, the decision that a VM(s) should be migrated is made by the DiProNN Control & Sessions Management module. This module monitors the actual node usage (in cooperation with the DiProNN Resource Management module), communicates with the other nodes in the network, and as soon as there is a necessity to redistribute a session's VM(s), asks the DiProNN Session Operator module(s) of the influenced session[12] to provide it. During the migration, the DiProNN Session Operator module(s) prepares a new set of forwarding rules (corresponding to the state after the migration) and once the migration finishes, applies them in cooperation with the Control modules of the relevant units[13].

Last, but not least, since the session, whose VM is being migrated, should be influenced as low as possible during the migration, the usage of a principle called *Live Migration* [64] is highly desirable. This enables copying of the migrated VM's content from one physical host to another without stopping it, which minimizes the time period, throught which the service/process running inside the VM is unavailable.

7.5 Session Termination

The termination of an established session could occur because of the user's request, the node administrator's request, or because of DiProNN's own decision. In the first case, the authorized user might ask the node for terminating a particular session by sending an appropriate DiCP request to the DiProNN Session Control port, while in the second one, the administrator(s) might ask the node for terminating an arbitrary session by sending the DiCP request to the DiProNN Control port. The termination, that is based on the node's decision, occurs in cases when a session breaks DiProNN rules in any way (for example, the Security module of a Processing Unit detects[14] a malicious AP, which may result in the immediate termination of the session that the AP belongs to).

In all the cases, the termination have to be performed by the DiProNN Session Operator module, which controls the particular session. Thus, if the termination is initiated by the node administrator or by the DiProNN itself, the DiProNN Control & Sessions Management module, which receives the request and solves the internal incidents, in-

[12]Note, that there cannot be running any APs belonging to different sessions inside a single VM—the definition of the APs/VMs mapping process disallows this (see the Section 7.3.1).

[13]Note, that the inter-node migration requires an application of similar session's transformation, as the one depicted in the Section 7.3 (especially, it requires opening new ports on the target's Distribution Unit in order to make the APs running in the migrated VM accessible).

[14]To detect malicious active programs, similar techniques, which are studied in the projects dealing with the intrusion detection systems, can be used—for example, the ones presented in [86, 125, 166].

structs the relevant DiProNN Session Operator module to terminate the session. If more DiProNN nodes participate on the session's processing, once a particular DiProNN Session Operator module becomes aware of the session's termination, it instructs all the DiProNN Session Operator modules controlling the session's subsessions to terminate them as well.

The (sub)session's termination is then performed on every involved node in the following way: the DiProNN Session Operator module sends the termination request to the Control modules of all the Processing Units, which run (sub)session's VMs, and to the Distribution and Aggregation Units. The units remove all the forwarding rules set for the particular (sub)session, and acknowledge the removing. After that, the DiProNN Session Operator module contacts the relevant VM/AP Management modules in order to stop and destroy the session's VMs; once destroyed, all the reserved resources are freed on the particular Processing Unit, and the stopping/destroying is acknowledged.

Sometimes, users might require the node to return data produced during the session's processing. In such a case, the relevant VM/AP Management modules are asked for saving the relevant session's VMs to a location dedicated on the Storage Unit, where the users might download it. Nevertheless, since the node may not want to provide the whole built-in EEs, inside which the session's standalone APs have run, the required files should be extracted from the VM in cooperation with the APM service and separately saved on the Storage Unit as well.

Finally, the DiProNN Session Operator module confirms session's termination to the user, closes the communication and ends its operation.

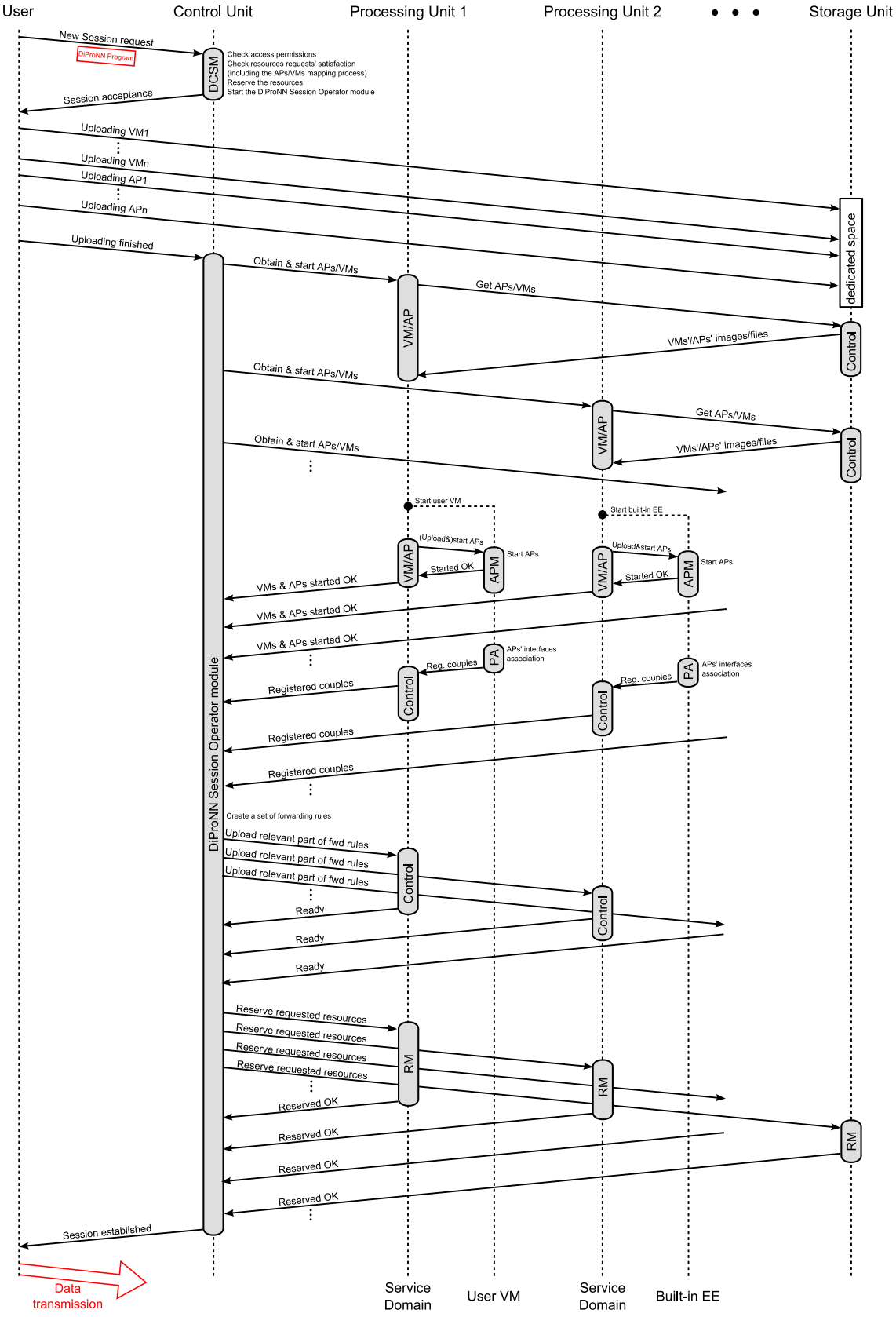


Figure 7.4: Sessions' establishment diagram.

Chapter 8

Distributed Sessions' Processing

The distributed processing depicted in the Section 7.3 allows several DiProNN nodes to participate on a single session's processing, so that more hardware resources are available to satisfy sessions' requirements. The session establishment process, which has been described in the previous chapter, is performed in a standard way—once the DiProNN Program describing the session, which should be established, is delivered to a DiProNN node, and once the node decides to distribute the session (i.e., there is no "better" node able to satisfy the whole session), the session is separated into multiple *subsessions* and the neighboring DiProNN nodes are asked to establish each of them (see the Section 7.3). Once all the subsessions are established, the original DiProNN Session is accepted (see an example of a distributed DiProNN Session in the Figure 8.1).

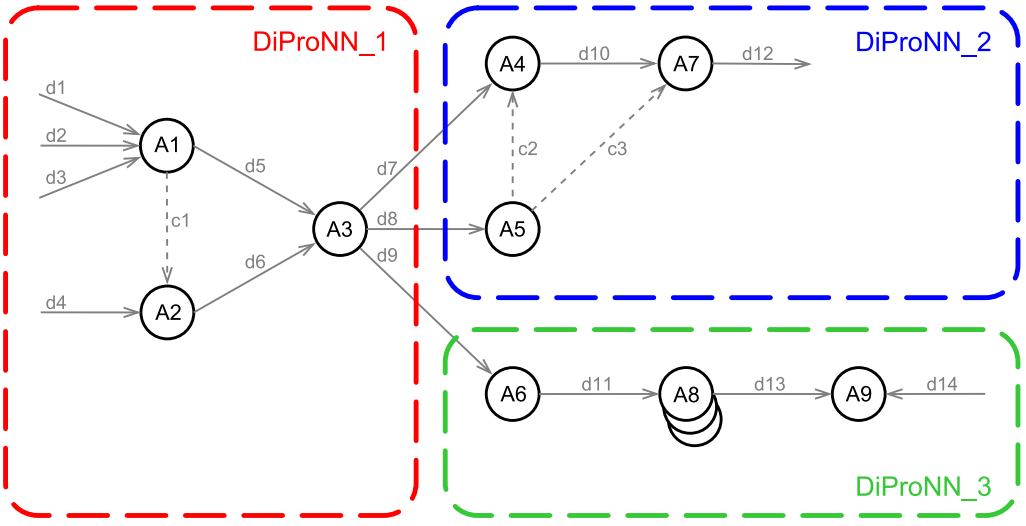


Figure 8.1: An example of a distributed DiProNN Session.

The whole session distribution process is under control of the entrance node (more precisely, under control of its DiProNN Control & Sessions Management module). Since (from the users' point of view) the distributed processing behaves as a normal processing provided by a single node, the users need not to notice it (if they are not informed by the node providing the session establishment)—the particular implementation might define

that all the data as well as control information is sent just to the entrance node[1], which manages all the necessary forwarding to relevant APs and/or another DiProNN nodes.

As obvious, the main task of the DiProNN node establishing such a distributed session, which should be performed in cooperation with an external service managing all the nodes in the network, is to find both a suitable separation of the particular DiProNN Session and a suitable mapping of all its APs over the Processing Units of appropriate DiProNN nodes (the APs/VMs mapping process). Let us denote this problem as the *DiProNN Scheduling Problem (DSP)*, which we formally describe together with several constraints it has to follow in the Section 8.1. The complexity of any DSP solution method is later studied in the Section 8.2, while a discussion of available scheduling techniques for the DSP follows in the Section 8.3.

8.1 Problem Description

This section formally describes the DSP. During the description as well as the rest of this chapter, we assume both the DiProNN Sessions and DiProNN nodes to be represented as graphs with nodes and edges (defined in the following subsections). Moreover, the constraints, which have to be satisfied by any DSP solution, are specified in the Section 8.1.3.

If not stated otherwise, in the rest of this chapter we suppose every virtual machine to contain just a single active program—as discussed previously, this enables precise resource requirements to be specified for every AP. Thus, in the following sections, let us omit the fact, that the AP is running in a VM, and let us assume both that the AP itself is able to request (and get) some amount of available resources and that the AP is directly able to be run on some Processing Unit.

Regarding the resource requirements, for simplicity reasons we focus just on the basic ones—the CPU (expressed, e.g., in cycles/s), amount of memory (e.g., in MB), amount of disk space (e.g., in MB), and requested link bandwidth (e.g., in Mbps). The DSP being aware of all the other resources (as mentioned in the Section 3.2.4) could be easily defined as an extension of the presented solution (especially, by adding necessary constraints similar to the ones defined in the Formulas 8.1—8.12).

8.1.1 DiProNN Session

Let us denote the set of all the DiProNN Sessions as S. Every DiProNN Session can be represented by the graph $s \in S, s = (A, D, C)$, consisting of nodes (A) and edges (D, C). Each node $a \in A$ then denotes a single AP, which requires several resources for its proper execution, such as CPU, RAM, and HDD (Processing Unit's and/or additional), represented by the values CPU_a, RAM_a, HDD_a, and $AHDD_a$ respectively. As mentioned before, every (non-parallelizable) AP is executed in one dedicated virtual machine, while the parallelizable APs are executed in several virtual machines[2], all of them having the same resource requirements.

Each graph's edge $d \in (D \cup C)$ denotes one communication channel between two active programs. There are two types of edges — D and C—representing data and con-

[1]Note, that (at least some of) the AP(s) receiving the stream from outside (the "first" AP(s) in the particular DiProNN Session) have to be running on the node asked for the session establishment—if the node is not able to satisfy at least a part of the session requested, the user is redirected to a different node, as described in the Section 7.3. Thus, the node does not perform packets forwarding to other DiProNN nodes only, which would result in unnecessary performance overhead.

[2]The number of virtual machines is equal to the `instances_count` variable set in DiProNN Program, or set dynamically by the Control Unit.

trol communication channels respectively. Every $d \in D$ also holds requested bandwidth $BAND_d$, representing the minimal amount of bandwidth necessary for the proper inter-node communication. The communication channels are further denoted as the *logical data edges* (the data channels—the set D) and *logical control edges* (the control channels—the set C) as well.

For example, the DiProNN Session example depicted in the Figure 8.1 could be described by the graph

$$s_{ex} = (\{a_1, a_2, a_3, a_4, a_5, a_6, a_7, a_8, a_9\}, \\ \{d_1, d_2, d_3, d_4, d_5, d_6, d_7, d_8, d_9, d_{10}, d_{11}, d_{12}, d_{13}, d_{14}\}, \{c_1, c_2, c_3\})$$

8.1.2 DiProNN node

Let us consider a couple (G, H), where the set G represents all the DiProNN nodes available in the network, and the set H denotes all the connection links (external interconnections) among them. Each $h \in H$ denotes a single physical connection link, providing limited bandwidth $BAND_h$, which denotes the maximum bandwidth available on this link. We kindly suppose, that each DiProNN node is directly accessible by all the others (e.g., the interconnection among the nodes is provided by the Internet).

Every DiProNN node $g \in G$ can be represented by the graph $g = (U, E, F)$. The graph's nodes represent the DiProNN units, where each unit $u \in U$ is either the Distribution Unit, the Aggregation Unit, the Control Unit, the Storage Unit, the Processing Unit, or the data/control interconnect switch. Let's define R as a set of all the Processing Units available in the particular DiProNN node g, such that $R \subset U$. Then, each $r \in R$ denotes a single Processing Unit (physical machine) having the following resources: CPU, RAM, and HDD, denoted as CPU_r, RAM_r and HDD_r. Moreover, the Storage Unit $t \in U$ could provides additional storage space, denoted as $AHDD_t$.

Since the DiProNN node assumes two types of interconnections—the data one and the control one—the graph assumes two types of connecting edges as well. First, each edge $e \in E$ denotes a single physical data interconnection link between two DiProNN Units, providing limited bandwidth $BAND_e$. Second, each edge $f \in F$ denotes a (low-latency) control interconnection link between two units. In this case, however, the communication bandwidth is not considered for any $f \in F$, since the control interconnection is used just for low-amount of small control messages only—it is supposed, that the available bandwidth will be always sufficient.

For example, regarding the DiProNN node depicted in the Figure 8.2, the set U includes eight DiProNN units (including the switches), while the set R contains just two of them (the Processing Units only). Regarding the interconnection links, since all of them—besides the Processing Unit's local ones provided by the VMM[3]—are supposed to be full-duplex, thus providing defined bandwidth independently in both directions, each link produces two graph edges (one for each direction) in the DiProNN node graph. Thus, the DiProNN depicted in the mentioned figure produces 12 data links in the set E and 14 control links in the set F.

[3] Although the virtual interconnection provided by the VMM appears to be full-duplex, its available bandwidth is usually influenced by its usage in both directions. These local interconnections are thus represented as single edges.

8.1.3 Constraints

In this section, we define several constraints that every scheduling technique must follow to build a valid DSP solution.

At first, let us assume an active program $a \in A$. The AP must be run (in a virtual machine) on a Processing Unit $r \in R$. The following constraints ensure, that all the resources requested by a must be available at r as well as additional storage requirements must be available at the Storage Unit t:

$$CPU_a \leq CPU_r \tag{8.1}$$
$$RAM_a \leq RAM_r \tag{8.2}$$
$$HDD_a \leq HDD_r \tag{8.3}$$
$$AHDD_a \leq AHDD_t \tag{8.4}$$

Moreover, if more APs are running on the same Processing Unit r, then the total requested resources must be less or equal to the capacity of r. Formally, if k APs $(a_1, .., a_k)$ are running on r, then

$$\sum_{i=1}^{k} CPU_{a_i} \leq CPU_r \tag{8.5}$$
$$\sum_{i=1}^{k} RAM_{a_i} \leq RAM_r \tag{8.6}$$
$$\sum_{i=1}^{k} HDD_{a_i} \leq HDD_r \tag{8.7}$$

Similarly, if more APs share the disk space $AHDD_t$ provided by the Storage Unit, the total requested additional storage resources must be less or equal to the capacity of t. Formally, if k APs $(g_{a_1}, .., g_{a_k})$ are running on the particular DiProNN node $g \in G$, then

$$\sum_{i=1}^{k} AHDD_{g_{a_i}} \leq AHDD_t \tag{8.8}$$

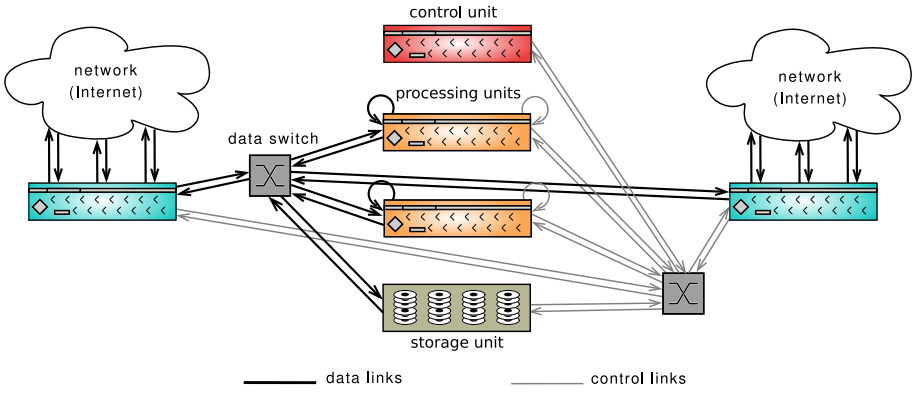


Figure 8.2: DiProNN node graph.

If a parallelizable AP requires k instances, the scheduler handles each of them as an independent AP, which allows to use the previously defined constraints. Still, one more constraint is necessary for the parallelizable APs: *all k instances of a single parallelizable AP must be mapped on the same DiProNN node* (e.g., to allow their fast communication via the control interconnection, and/or to allow simpler management of the number of instances in the case of dynamic adaptations to processing conditions, etc.).

Similar rules apply concerning the communication bandwidth. First, let us assume the simplest scenarios: if two APs (or a single AP with the Storage Unit) communicate over a logical data edge $d \in D$ using a single physical data interconnection link $e \in (E \cup H)$, then

$$BAND_d \quad \leq \quad BAND_e \tag{8.9}$$

Again, if a single physical interconnection link e is shared by k logical data edges $(d_1, .., d_k)$, the overall requested bandwidth must be less or equal to the bandwidth available on the link, i.e.,

$$\sum_{i=1}^{k} BAND_{d_i} \quad \leq \quad BAND_e \tag{8.10}$$

However, in real situations, the active programs (AP_1, AP_2), which communicate over a logical data edge d, are usually placed in different DiProNN Processing Units or in different DiProNN nodes. Then, the logical data edge d will be mapped onto several physical interconnection links, which will create a path from AP_1 to AP_2. Let us assume that d will be mapped onto l different physical interconnection links $(e_1, ..., e_l \in (E \cup H))$ providing different bandwidths $(BAND_{e_1}, ..., BAND_{e_l})$. Then, the requested bandwidth must be less or equal to the minimal bandwidth available on the path:

$$BAND_d \quad \leq \quad \min_{1 \leq i \leq l} BAND_{e_i} \tag{8.11}$$

Also, each of the l physical interconnection links might be simultaneously used by different APs, so that the $BAND_{e_i}$ has to be shared by multiple logical data edges. Let us assume, that for given i the physical interconnection link e_i is going to be used by k logical data edges $(d_{1,i}, ..., d_{k,i})$. Then, the following constraint has to apply for every $i = (1, ..., l)$ and corresponding k:

$$\sum_{j=1}^{k} BAND_{d_{j,i}} \quad \leq \quad BAND_{e_i} \tag{8.12}$$

Moreover, all the APs, that communicate with each other via the control interconnection, *must be placed within the same DiProNN node*. However, on the other hand, two APs, that communicate over the data interconnection only, can be placed in different DiProNN nodes, if the previously defined constraints are satisfied.

Finally, there are two another constraints, that have to be satisfied. First, the scheduler has to ensure, that there are no redundant schedules generated (to avoid redundant mappings of an active program on several Processing Units). And second, the scheduler has to guarantee that every DiProNN Session will be mapped as a whole—either all the active programs belonging to it are mapped, or none of them.

Let there be k DiProNN Sessions in S and l DiProNN nodes in G. Let $x_{ra} = 1$ if the AP a was successfully mapped on the Processing Unit r, and $x_{ra} = 0$ otherwise.

$$A = \bigcup_{s=1}^{k} A_s$$
$$R = \bigcup_{g=1}^{l} R_g$$
$$\forall a \in A : \sum_{r \in R} x_{ra} \leq 1 \tag{8.13}$$
$$\forall s \in S : \quad \sum_{a \in A_s} \sum_{r \in R} x_{ra} = 0 \ \vee \ \sum_{a \in A_s} \sum_{r \in R} x_{ra} = |A_s| \tag{8.14}$$

The Formula 8.13 then provides a new constraint guaranteeing that any AP a from any DiProNN Session $s \in S$ will be mapped on at most one Processing Unit r of all the available DiProNN nodes (defined by the set G). The latter Formula 8.14 further ensures, that either all or none the APs $a \in A_s$, which belong to a session s, must be mapped on some Processing Unit $r \in R$ (note, that the set R contains all the Processing Units of all the DiProNN nodes in the network).

8.2 DSP Complexity Analysis

In the previous section, the formal description of the considered scheduling problem, which involves mapping of the DiProNN Sessions on the DiProNN node(s), was presented. All the important parameters required for the proper scheduling were described, including all the related constraints that have to be satisfied. The scheduler takes a DiProNN Session description (the DiProNN Program or the DiProNN Session Graph) specified by the user and tries to map it on the available DiProNN node(s) (in fact, it maps session's APs to available nodes' Processing Units), with respect to presented constraints.

In this section, we discuss the complexity of such a solution technique. At first, we define an objective function, which helps us to measure the quality of a particular scheduling solution found, and which enables us to compare all the solutions so that we are able to choose the best one. Later, we study the complexity of the DSP problem by transforming a well-known problem, whose complexity is widely-known and already proved, to it.

8.2.1 Applied Objective Function

Before proceeding to an analysis of the DiProNN Scheduling Problem complexity, we define an objective function helping us to find the optimal scheduling solution. The objective function adopted represents a real-life-based function heading towards maximizing the utilization of the DiProNN nodes in the network.

The objective function assumes a set G of available DiProNN nodes, each node $g \in G$ having l_g Processing Units. Further, a set S of DiProNN Sessions has to be given, each session $s \in S$ having k_s active programs. The function also considers the active programs' profit values $p_a, p_a = 1, ..., MAX$ for every active program a. For most real-life purposes, the profits will be equal to the APs' particular resource requirements (e.g., CPU time—thus measuring the CPU utilization of all the DiProNN nodes) or a value representing a combination of all the requested resources. However, the profits might also cover an additional information—for example, it can be a function of the resource requirements together with the particular session's estimated running time and/or session's owner priority.

The resulting objective function is then defined as

$$\begin{aligned} F &= \sum_{g \in G} \sum_{r_g=1}^{l_g} \sum_{s \in S} \sum_{a_s=1}^{k_s} p_{a_s} x_{r_g a_s} \\ x_{r_g a_s} &\in \{0,1\} \end{aligned} \tag{8.15}$$

where $x_{r_g a_s} = 1$ if the active program a belonging to the DiProNN Session s was successfully mapped on the Processing Unit r of the DiProNN node g, and $x_{r_g a_s} = 0$ otherwise.

Clearly, the function F returns a weighted sum of successfully mapped DiProNN Sessions satisfying all the constraints mentioned in the Section 8.1.3. The goal of the DSP scheduler is to find a mapping which provides the maximal F.

8.2.2 The Proof of the DSP Complexity

To prove the DSP complexity let us consider a *relaxed DSP* problem involving just a single DiProNN node (including several Processing Units) and simplified DiProNN Sessions, consisting of just a single active program.

During the proof, we use the *Reduction* technique [96, 103]—a transformation of one problem (A) into another problem (B) in the way, that solutions to B exist and give solutions to A whenever A has solutions. In other words, if A reduces to B ($A \leq_T B$, where T is a type of the transformation) means, that the problem A is no harder than B, and B is no easier than A. Depending on the transformation used, this technique can be used to define complexity classes on a set of problems.

Thus, using the simplified DSP problem we reduce[4] a special case of the well-known *0-1 Multiple Knapsack Problem (MKP)* [183, 218], which is known to be $\mathcal{NP}$-hard [183], on the relaxed DSP problem, thus showing that the complexity of the relaxed DSP is $\mathcal{NP}$-hard. Finally, because any instance of the relaxed DSP can be reduced to an instance of the original DSP, we show that the original DSP is $\mathcal{NP}$-hard as well.

0-1 Multiple Knapsack Problem

The 0-1 MKP—one of the knapsack problems, that belong to most widely studied problems in the Combinatorial Optimization—describes the following problem: there are k items that should be packed into l knapsacks of distinct capacities c_i, $i = 1, ..., l$. Each item j has an associated profit p_j and weight w_j, and the problem is to select l disjointed subsets of items, such that each subset i fits into the capacity c_i and the total profit of all the selected items is maximized. Thus, the 0-1 MKP can be formally defined as [183]

$$\text{maximize} \quad \sum_{i=1}^{l} \sum_{j=1}^{k} p_j x_{ij} \tag{8.16}$$

$$\text{subject to} \quad \sum_{j=1}^{k} w_j x_{ij} \leq c_i, \quad i = 1, ..., l, \tag{8.17}$$

$$\sum_{i=1}^{l} x_{ij} \leq 1, \quad j = 1, ..., k, \tag{8.18}$$

$$x_{ij} \in \{0,1\}, \quad i = 1, ..., l, \quad j = 1, ..., k,$$

[4] During the proof we use the *Linear-time reduction* [96].

where $x_{ij} = 1$ if the item j is assigned to the knapsack i, and $x_{ij} = 0$ otherwise. Usually, the coefficients p_j, w_j and c_i are assumed to be positive integers, and to avoid trivial cases it is demanded that

$$\max_{j=1,...,k} w_j \leq \max_{i=1,...,l} c_i \qquad \min_{i=1,...,l} c_i \geq \min_{j=1,...,k} w_j \qquad \sum_{j=1}^{k} w_j > \max_{i=1,...,l} c_i. \tag{8.19}$$

The first assumption ensures that each item j is able to be fitted into at least one knapsack, otherwise it should be removed from the problem. Second inequality ensures that none knapsack is smaller than the smallest item. Finally the last inequality avoids trivial solution where all the items fit into the largest knapsack. This 0-1 MKP is known to be $\mathcal{NP}$-hard [183].

However, in the DiProNN, the profits have to be related to the weights (in fact, resource requirements)—thus, by maximizing the profits we maximize the utilization of the nodes, which is the desired scheduler's behavior. Nevertheless, the special case of the 0-1 MKP, where the profits and weights are somehow related (in our case identical[5]), still remains too hard, and thus belongs to the class of $\mathcal{NP}$-hard problems as well [183].

In the rest of this section, our goal is to show that the special case of the 0-1 MKP can be represented as the relaxed DSP, which will prove that the relaxed DSP is $\mathcal{NP}$-hard as well (remember, $MKP \leq_T DSP_{rel}$ means that the MKP is not harder than the relaxed DSP).

Relaxed DSP

Let us assume the *relaxed DSP* with a single DiProNN node containing l Processing Units $r, r = 1, ..., l$ having different CPU_r values. Further, let there be k DiProNN Sessions to be scheduled—each session is being composed of just a single AP (it represents the simplest version of the DiProNN Sessions). Thus, we have k active programs $a, a = 1, ..., k$ to be scheduled on the node's Processing Units.

Further, let us assume that every DiProNN Session (in fact, the active program) has its profit equal to the amount of computational resources requested (as defined in the objective function), specified by the CPU_a parameter. Other constraints, such as the RAM_a, HDD_a, and the $AHDD_a$ as well as the requested communication bandwidth will be ignored[6].

As mentioned previously, the relaxed DSP's objective is to maximize a single DiProNN node's utilization by mapping given DiProNN Sessions (active programs) on the node's Processing Units, with respect to AP's profits (once again, equal to their CPU require-

[5]This modified problem is also very similar to the *Multiple Subset Sum Problem (MSSP)* [53], which however differs in an assumption, that all the knapsacks' capacities are identical (thus, the MSSP can be considered as a special case of the modified 0-1 MKP). Similarly to the 0-1 MKP and modified 0-1 MKP problems, the MSSP is $\mathcal{NP}$-hard as well [53].

[6]This is equivalent to setting $RAM_a = 0$, $HDD_a = 0$, $AHDD_a = 0$, and $BAND_d = 0$ for all the requested data connections $d \in D$. Then, all the relevant constraints hold trivially—any solution of the original problem will be also a valid solution of the relaxed DSP.

ments). Therefore the relaxed DSP can be formulated as:

$$\text{maximize} \quad \sum_{r=1}^{l}\sum_{a=1}^{k} CPU_a x_{ra} \tag{8.20}$$

$$\text{subject to} \quad \sum_{a=1}^{k} CPU_a x_{ra} \leq CPU_r, \quad r = 1, ..., l, \tag{8.21}$$

$$\sum_{r=1}^{l} x_{ra} \leq 1, \quad a = 1, ..., k, \tag{8.22}$$

$$x_{ra} \in \{0, 1\}, \quad i = 1, ..., l, \quad j = 1, ..., k,$$

where $x_{ra} = 1$ if the active program a is assigned on a Processing Unit r, and $x_{ra} = 0$ otherwise.

As obvious, the Formula 8.20 is adopted from the objective function (Formula 8.15), where the set G contains just a single DiProNN node, each DiProNN Session contains just a single active program, and each active program's profit is equal to the amount of CPU resources it requires. Then, the Formula 8.21 covers both the constraints 8.1 and 8.5, while the Formula 8.22 is equal to the constraint 8.13. The constraint 8.14 holds trivially, since every session contains just a single AP.

Further, we can assume that the coefficients CPU_a and CPU_r are positive integers, and to avoid trivial cases we demand that

$$\max_{a=1,...,k} CPU_a \leq \max_{r=1,...,l} CPU_r \quad \min_{r=1,...,l} CPU_r \geq \min_{a=1,...,k} CPU_a \tag{8.23}$$

$$\sum_{a=1}^{k} CPU_a > \max_{r=1,...,l} CPU_r \tag{8.24}$$

Analysis Discussion

The analysis shows, that we can formulate the $\mathcal{NP}$-hard 0-1 MKP problem as the relaxed DSP problem. The applied transformation is linear involving just variables renaming. Moreover, for any instance of the relaxed DSP there is a simple linear reduction, which generates an original DSP instance ($\forall a \in \{1, ..., k\} : RAM_a = 0$, $HDD_a = 0$, $AHDD_a = 0$, $\forall d \in D : BAND_d = 0$). Thus, any algorithm solving the original DSP would also solve such newly generated instance and the resulting solution would be also the valid solution for the relaxed DSP. Therefore, finding the optimal solution with respect to the applied objective function for the original DSP is also the $\mathcal{NP}$-hard problem.

In fact, the original DSP is much more complicated than the relaxed DSP due to additional constraints such as $RAM_a, HDD_a, AHDD_a$, and/or $BAND_d$. Also, the DiProNN Sessions are usually composed of several APs communicating with each other, and could be generally spread over several DiProNN nodes, where they are mapped on particular Processing Units. This involves additional constraints that have to be satisfied, as shown in the Section 8.1.3.

8.3 DSP Scheduler Discussion

The fact, that the DSP belongs to the set of $\mathcal{NP}$-hard problems implies that as long as it is not proven that $\mathcal{P} = \mathcal{NP}$[7], one cannot expect to have a polynomial-time algorithm for solving it. Since an algorithm solving the DSP needs to provide sufficiently high performance to be useful in practice—DiProNN Sessions' configurations should be constructed in the real-time during the establishment time or migration decisions—its complexity should be addressed by some approximation techniques.

In the literature, there are several works addressing similar placement techniques for component-based distributed applications processing data streams. Basically, these techniques could be divided into two categories—the dynamic components' selection and deployment techniques, and the deployment techniques only, which are more suitable for the DSP scheduling. Thus, instead of proposing a new scheduling technique we show and discuss some existing ones, which the DSP scheduler could be based on.

8.3.1 Components' Dynamic Selection & Deployment Scheduling Techniques

The dynamic component selection and deployment techniques allow distributed applications to flexibly and dynamically adapt the processing workflows to situations in both current resource utilizations and clients' demands. For example, once an available network bandwidth lowers, the planning algorithm in cooperation with a particular framework might decide to change the processing workflow in the way that all the transmissions will be compressed. This can lower the bandwidth required by the data streams without any undesirable impacts on the application itself (obviously, if an increase of the end-to-end latency does not matter).

Such planning techniques are studied by many authors. For example, in [151, 152] the authors propose a general model and discuss techniques usable for such an application configuration problem, and present an AI planning-based [67] algorithm (called *Sekitei*) for solving it. The *Pegasus* [31] is another planner employing AI techniques to generate grid workflows (application configurations) to achieve a user objective. The planning of single-input and single-output components could be further performed using the *Conductor* framework [286], which allows composable and transparent applications' adaptations to unfavorable network characteristics, while multiple-inputs and multiple-outputs components could be planned by the *Ninja* planning module [106]. And there are many other works [38, 89, 93, 132, 229] focusing on similar objectives.

However, these scheduling algorithms are not straightforwardly usable in the DiProNN, since besides considering current resources' state for placing the components they also focus on flexible workflow assembling to suit both network environment conditions and users' needs. Nevertheless, the second category—represented by pure deployment scheduling techniques—is more suitable for the DSP scheduler and thus is discussed in more detail in the following section.

8.3.2 Components' Deployment Scheduling Techniques

The pure deployment scheduling techniques assume that there is given a particular processing workflow, which consists of several components, and the goal is just to place the

[7] The relationship between the complexity classes $\mathcal{P}$ and $\mathcal{NP}$ is an unsolved question in the theoretical computer science [246]. It is considered to be the most important problem in the field, since the determination of the status of this question would have dramatic consequences for the potential speed of solving many difficult and important problems.

components on an existing network infrastructure to meet their resource requirements (without any workflow's modifications to adapt it to current network conditions). In spite of the fact, that there have been proposed many techniques, which focus on common distributed component-based applications' scheduling, most of them are not usable for the DSP scheduler since they do not fit the stream-processing applications' fundamental characteristics—the continuous data processing and unknown components' duration time. Thus, just a few of them are discussed here.

An example of a suitable scheduler for the component-based stream-processing applications is the *Streamline* [4] scheduling heuristic, which is designed over an existing grid framework using the Globus Toolkit [87]. The Streamline uses the Simulated Annealing technique [164] as an approximation for an optimal schedule, which is infeasible to implement except for very small applications. It supposes the streaming application to be represented by a directed acyclic dataflow graph (DAG) consisting of nodes (continuously running applications) and edges (data links between applications). The Streamline considers the static information about available resources (e.g., machine architecture, CPU speed, amount of memory, and hardware configuration), the dynamic information of available resources and communication capabilities in the target environment (such as an estimate of available processing cycles or the end-to-end network bandwidth), and different applications' and resources' specific policies.

The *MediaNet* project [119] proposes a two-level scheduling service for the component-based processing applications—in addition to using a *local scheduling* only (scheduling on an individual node), the MediaNet also employs a *global scheduling service* to divide tasks and flows among network components. Similarly to the Streamline, the applications are also supposed to be represented as DAGs and the target network infrastructure as a graph with nodes (computing elements) and edges (data links). As depicted, the scheduling service assumes an existence of a global scheduler, which computes a session subgraph for each node and sends it to the local scheduler running on each node. The local schedulers implement the sessions' subgraphs and periodically report the local resources' usage to the global scheduler, which can periodically recompute and redistribute its schedules as well. Similarly to the works belonging to the first category, the MediaNet scheduling service allows adaptations to current network conditions as well. However, these adaptations are not determined by the system, but by the end-users—each user contributes with a list of alternative specifications and associates a utility value with each of them. The primary goal of the MediaNet's schedulers is to maximize each user's utility.

In [262], the authors propose two resource allocation heuristic algorithms for processing streams on computational grids. Both algorithms are based on the market mechanisms [66]; one uses a centralized market (the *Single Market Resource Reservation System* algorithm) and the other decentralized markets (the *Multiple Market Distributed Resource Reservation System* algorithm). The former one uses a variation of the first-fit-decreasing-value heuristic to assign tasks to machines—the tasks are sorted in a decreasing order of utility and assigned using the best-fit-decreasing-value heuristic to target machines, where local optimizations further take place. Against it, the latter one assumes multiple independent markets (one for each machine resource)—initially, the stream applications are assigned randomly to balance the number of streams per machine. Once the applications are assigned, a central agent pairs up a below-average priced machine with an above-average priced machine and the paired machines move streams from one to another to increase the sum of their utilities.

Unfortunately, the described approaches do not take into account the density of an communication among the components, which is highly beneficial for the DSP sched-

uler. Since the components with a dense interaction should be placed on the same nodes, the presented algorithms should be complemented by the technique presented in [289], where the authors propose a general solution to the interaction aspect of the component placement problem. The basic idea behind such a technique, which considers both interaction and non-interaction properties of the system, is that there is an interaction model of the system created (modelled using the Component-interaction automata language [41]) and a set of candidate component placements is computed using an existing, non-interaction algorithm (to enable the integration with the existing solutions solving the non-interaction aspect of the component placement problem). Once finished, the candidate placements are evaluated depending on the frequency of inter-component communication and the most optimal component placement is selected.

8.3.3 DSP Scheduler Conclusions

Even thought the DiProNN Sessions are represented as directed graphs as well, one could notice, that they cannot be directly used as the inputs for the scheduling algorithms discussed in the previous section. The reasons are, that the DSP scheduler has to cope with two kinds of the communication channels (the data and control ones) between the components (i.e., APs in the case of the DiProNN Sessions) and that the DiProNN Sessions might contain parallelizable APs, which the discussed algorithms do not support. Thus, the DiProNN Sessions have to be transformed in a way, which makes them applicable for the algorithms' inputs, but which does not affect the scheduling decisions performed by the algorithms (in other words, the scheduling decisions for both the original and the transformed sessions must be the same). This can make the existing scheduling solutions suitable for solving the DSP.

Regarding the parallelizable APs, these could be—together with their input and output communication channels—easily substituted by their simple multiplication in the amount requested (if known) or in a defined initial amount. The bandwidth requirements are divided among all the instances, while the other resources' requirements are applied for every parallel instance in the amount requested by the AP.

Regarding the problem of the two communication channels' kinds, there are two transformations, that come into the mind at first. The first one is to declare both kinds of the communication channels as a single one for the algorithms—i.e., to consider both the data and control channels as a single kind—and to let them compute a suitable placement(s) for such a transformed session. However, such a computation does not need to be always feasible, since this transformation can easily break the single presumption the algorithms have—that is the *acyclic* form of the sessions' graphs.

The second possible transformation is to omit the control communication channels at all. At first note, that such a transformation cannot affect the DSP's scheduling decisions in any way: as depicted in the beginning of this chapter, there are no resource constraints defined for the control interconnections—they provide an "unlimited" bandwidth for all the control messages. Thus, the scheduler does not have to take available control links' bandwidths into account and can place the particular APs on an appropriate Processing Unit(s) just according to their resource requirements.

Omitting the control communication channels can ensure the acyclic form of the sessions' graphs[8], but cannot ensure the other requirement—that every two active programs communicating via a particular control communication channel will be placed on the same DiProNN node. As stated in the beginning of this chapter, placing the APs,

[8]We assume that the data paths of the DiProNN Sessions will not contain cycles.

which communicate via a (low-latency) control interconnect, on different DiProNN nodes breaks the main purposes of such an interconnection, since it is intended to provide abilities for as fast as possible communication and synchronization of the APs.

However, to make such a DSP scheduler able to generate placements satisfying this requirement, the "omitting control channels" transformation has to be extended by a definition of some application scheduling policies applied for the schedulers (e.g., as the ones used in the cases of the Streamline and MediaNet schedulers discussed before). These policies can ensure, that every two components communicating via a particular control communication channel as well as that all the instances of a parallelizable AP will be placed on the same DiProNN node (see the constraints in the Section 8.1.3).

Chapter 9

Further DiProNN's Features

9.1 Quality of Service Support

The ability of a network (in fact, its network nodes) to provide a different priority to different applications, users, or data flows, or to guarantee a certain level of performance to a data flow is called *Quality of Service (QoS)*. For example, a required bit rate, delay, jitter, and/or packet dropping probability may be guaranteed in common network nodes. The QoS guarantees are primarily requested by real-time streaming (multimedia) applications, since these are often delay sensitive and require the network to provide specific performance parameters. Such guarantees are important especially in situations, when the network capacity is insufficient—in the absence of network congestion, QoS mechanisms are not (usually) necessary.

The QoS approaches, that are used in common computer networks, enforce certain parameters (e.g., a queueing strategy, a priority), which are not sufficient for the Active/Programmable networks. The reason is, that such networks enable the users to run programs on the inner network nodes and thus other computing parameters (e.g., processor time, amount of free memory) have to be guaranteed as well.

In the previous chapters, there has been supposed a resource management system—the DiProNN Resource Management module cooperating with the Resource Management modules of the relevant units—which is able to reserve intended resources for particular VMs and communication channels. However, such a standalone system is not sufficient for providing the QoS guarantees, since the resource delegations must be supported by accurate and powerful resource schedulers (provided by the particular virtualization system), so that the node might become able to guarantee requested resources to the sessions. However, since the DiProNN does not rely on a specific virtualization system, the rest of this section describes and defines general requirements, that the employed resource schedulers have to satisfy.

9.1.1 Schedulers

As apparent, the scheduling algorithms are very important part of the RMS and its ability to provide QoS guarantees, because they affect both an overall performance and keep all the required resources in desired limits. Since resource characteristics vary, the scheduling algorithms must be designed in a resource specific manner. For example, CPU context switching is more expensive compared to switching between flows in the case of network scheduling [100]. Therefore, the efficiency of the CPU scheduling improves, if virtual machines can receive a minimum CPU quantum before being preempted. Disk scheduling,

unlike both the CPU and network scheduling, must consider request locations to limit seek times and rotational latency overheads[1], memory schedulers, in order to match actual memory use, must estimate the current working set of active programs. Therefore, all the schedulers must examine relevant resource states (e.g., disk state, whether it is spinning or parked) in addition to the QoS specifications.

Furthermore, the resource requirements are often related to each other. For example, when requesting high network bandwidth while having only a small amount of the CPU time, it is not possible to reach required bandwidth, because there is an insufficient CPU time to send all the packets. Thus, the scheduling algorithm's design must be sophisticated enough to take such inter-dependencies into account.

Except mentioned, all the resource schedulers have to satisfy the following five requirements[2]:

- *Admission criteria*—the admission of a new virtual machine should not infringe the resource guarantees given to the virtual machines already running. In the case of a conflict possibility, necessary steps need to be taken like re-mapping process or rejecting the relevant DiProNN session to avoid VMs' affections.
- *Real-time guarantees*—the design of the scheduling algorithms must satisfy real-time constraints in terms of ensuring guaranteed scheduling for each virtual machine within their jitter bounds, if any.
- *Fairness criteria*—it should be possible to schedule both types of virtual machines competing for a shared resource—the ones without any resource requirements as well as the ones requiring some resource guarantees. If a non-reserved part of the particular resource exists, the lower priority non-guaranteed VMs should not be completely starved out of the resource by higher priority tasks corresponding to the guaranteed services.
- *Maintenance and policing criteria*—the policing criteria requires to ensure that the resource-consuming VMs do not infringe the resource guarantees of other VMs competing for the resource. The maintenance criteria imply setting up re-mapping process or dropping further requests in the case of a resource overload condition.
- *Throughput criteria*—the scheduling policy of the particular scheduler should be able to schedule as many virtual machines as possible.

9.2 Hardware Support

As already mentioned in the motivation chapter, besides the benefits already mentioned, the virtual machines also bring some drawbacks once being employed in the network nodes. These drawbacks are mainly related to a performance overhead necessary for VMs' management, which is visible especially for I/O operations (see the tests presented in the Section 12.2.3)—every I/O must be intervened by the Service Domain of the particular virtualization system [179]. Since the DiProNN is performing lots of network I/O

[1] This fact is not entirely true for the Solid state disks (SSD), whose seek times are extremely low and more or less constant, and which do have zero rotational latency.

[2] Since the DiProNN's QoS assurance is closely related to multimedia applications, the requirements laid on the scheduling algorithms are very similar as the requirements in multimedia operating systems, that are defined in [100].

operations (it has to receive data, that should be processed, and send them out of the node after the processing), this overhead can significantly influence its overall performance.

Thus, in cooperation with the Liberouter project[3] of the Cesnet association[4], we have sketched a FPGA-based programmable hardware network card accelerating both the DiProNN's forwarding mechanism and the whole network stack, with which could be further each VM provided. Before we present the sketched card's architecture and its benefits, we describe the NetCOPE platform that the card is based on.

9.2.1 NetCOPE Platform

The NetCOPE [184], which has been designed by the Liberouter research group, is intended as a computing platform accelerating analyses of high-speed network traffics performed on the commodity hardware. The NetCOPE enables a computing task, that is required to be performed on the incoming data, to be split between the NetCOPE and the system processor(s)—while the NetCOPE performs data preprocessing in hardware (e.g., headers' parsing, packet payloads' analyses, etc.), the system processor(s) computes just highly-specific user-level functions. Such a processing combination accelerates the whole computing a lot, since all the data need not to be transmitted through the system buses to the system processor(s), where they should be computed, and sent outside the node throught the buses as well (if necessary). This also saves the cycles of the system processor(s), making such a system being able to process amounts of data, whose processing would be otherwise infeasible on the same system without the NetCOPE.

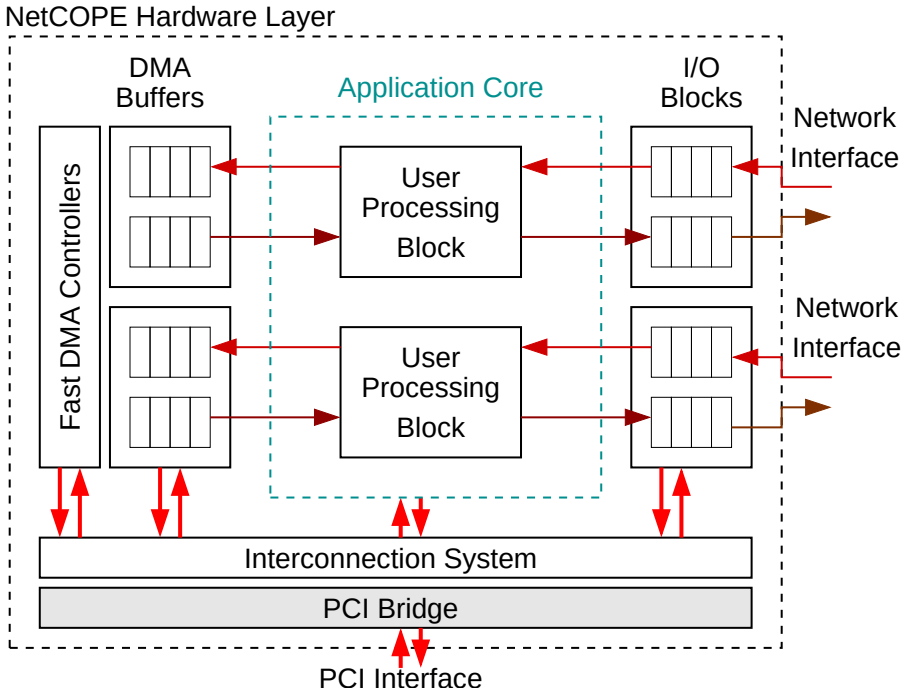


Figure 9.1: The NetCOPE hardware architecture [184].

The platform, which is depicted in the Figure 9.1, consists of the I/O blocks used for receiving and transmitting the incoming data via the Ethernet protocol, the DMA buffers

[3] http://www.liberouter.org/
[4] http://www.cesnet.cz

for fast data transfers between the adapter and the host RAM, the DMA controllers, and an interconnection system providing the communication between components placed in the FPGA and a system bus (PCI, PCI-X or PCI Express) [184]. A data packet, that arrives at a network interface, is passed through the I/O blocks to the *User processing blocks*, where the user-defined (pre)processing takes place. The results of the user processing are then stored into DMA buffers and further transferred to a software part of the processing application. Similarly, once processed in the software, the data are passed directly to the I/O blocks and through the network interfaces sent away from the node.

9.2.2 DiProNN HW-accelerated Network Card

As already mentioned, the DiProNN HW-accelerated network card is based on the NetCOPE platform, where a single User processing block provides the packets' forwarding among the APs running on the particular unit. The card architecture is depicted in the Figure 9.2—it consists of the *RX* (receive) and *TX* (transmit) I/O buffers, the processing block performing Netfilter-like packet headers modifications (the forwarding mechanism), and the Host interface (the Interconnection System, the PCI Bridge and the PCI Interface).

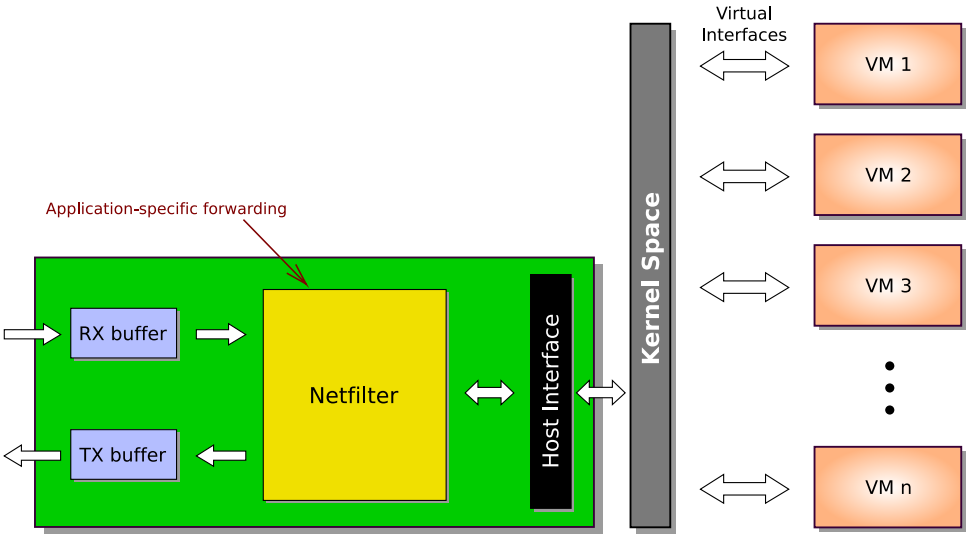


Figure 9.2: Schema of the DiProNN HW-accelerated Network Card Architecture.

In the Service Domain's kernel space, there is a driver controlling the card behavior and providing two basic functionalities. First, it sets the forwarding rules in the processing block based on the information from the Control module of the particular Processing Unit[5]—the forwarding rules then define the packets' forwarding performed by the card. And second, the driver ensures the forwarding of the packets, that come from and are destined to the virtual machines, which are located on the same unit—this does not require the packets to pass through the system buses in cases, when both the sending and receiving APs are located on the same unit.

[5]Even thought primarily designed to the Processing Units, these acceleration card could be naturally used in the Distribution and Aggregation Units as well. However, since these units perform just a simple packet forwarding and do not run a virtualization system, the performance improvement will not be so significant as in the case of employing it in the Processing Units.

The Processing Units, that should run multiple virtual machines, could benefit from yet another feature, that the card offers. Since the card's driver is able to emulate multiple network interfaces, the unit's Service Domain can dedicate each of them to a particular virtual machine. This enables the incoming/outcoming packets to bypass the Service Domain without any modifications of the VMM[6] or the guest VMs. Thus, the Service Domain need not intervene every VMs' network I/O operation, while the forwarding capability, that is necessary for ensuring proper data flows among the APs, remains unaffected (except being accelerated in hardware).

Even thought the card is primarily considered for accelerating the data packets' manipulations only, the acceleration of control packets' manipulations could be also performed in a similar way. Nevertheless, in such a case, the control interconnect has to be realized using Gigabit or 10 Gigabit Ethernet interconnect, since the NetCOPE is currently unable to make use of a highly-specialized interconnects (like the Myrinet or the Infiniband).

[6]Except the ones related to the card driver, since this is supposed to support the card.

Chapter 10

Applications

In this chapter, we present several possible applications of the proposed node. There is a variety of useful network services that can benefit from a data processing at intermediate network nodes, enabling the end hosts to deliver a high performance (from users' point of view) on common PC systems—the data could be (pre)computed on the inner network nodes, while the end host(s) can focus just on a (visual) presentation of the computed data without wasting their performance on the computations. In cases of more data senders/receivers, the data (pre)computations performed inside the network can further utilize the network links' capacities in a more efficient way by gathering/scattering the data on a strategic network points closer to data senders/receivers. Again, from users' point of view, all these services can lead to better end-to-end performance delivered by the networking system.

The list of possible applications, that is presented in this chapter, is not exhaustive—since the node supports uploading of user applications, it might be used for almost every data processing[1] one can imagine. It is "only" necessary to compose such a processing application and an appropriate DiProNN Program, and upload it into the DiProNN. Thus, just a few examples of applications, which the proposed node could be used for, are mentioned in the following sections.

10.1 High-quality Video Presentations, Demos, and Lectures

Remote presentations, demos, and lectures delivered using video streams are an efficient way of delivering pieces of information to many people throughout the world. Scientists are able to share their knowledge, experiences, and ideas with large and wide variety of people without any needs to meet on a single place in a specified and restricted time, which would have further resulted in loosing the time of all the parties by travelling. The remote collaboration enables unique technologies, materials, inventions, things, working procedures, surgeries, etc. to be shared and provided to many people in the real-time, and thus the general knowledge in the particular area can be increased. However, usually a very high-quality video is necessary to provide sufficient details of the content delivered (at least to some critical end hosts).

An example of such a high-quality video can be an uncompressed HD video stream, which takes approximately 1,5 Gbps of the network bandwidth, or so-called 4K video stream consisting of four independent uncompressed HD streams (and thus taking approximately 6 Gbps of the network bandwidth). Besides delivering these streams to ac-

[1] However, not only passing data processing, as depicted in the Section 10.7.

cordingly equipped end host(s), it might be desired to distribute them to multiple clients, even the ones not having sufficient network connectivity. Thus, to make the streams accessible also for clients with lower bandwidth connectivity, a video down-sampling becomes necessary. For example, in the case of distant surgeries, the real-time video has to be provided to a remote surgeon in the highest quality possible, while a down-sampled real-time video could be provided to medical students in lecture rooms, even if they do not offer sufficient connectivity to present the original, high-quality video.

Such multicast-like stream distribution and down-sampling can be provided using the DiProNN nodes inside the network—see the very similar application example described in the Section 13.1. If a client is able to specify its transcoding requirements, the down-sampling can be further performed on a per-client basis, depending on the user's real network link capacity.

Figure 10.1: A High Performance Computing lecture taught at the Louisiana State University in Baton Rouge (USA), which has been delivered to the Masaryk University in Brno using an uncompressed HD video stream and displayed on the SAGE tiled displays [203].

10.2 High-quality Videoconferences and Multimedia Stream Compositions

The high-quality videoconferencing is an efficient tool for an interactive scientific collaboration in the research community, especially for researchers separated by a wide distance. In addition to a visual communication with remote collaborators, the participants of such a conference can exchange complex high resolution images, e.g., satellite images, computer simulations, microscope views, and complex medical images.

As opposed to the previous section, which focuses on a content delivery from a single point of interest to multiple clients/auditors (so-called *1:N* communication model), dur-

ing videoconferences the content is delivered from multiple senders to multiple clients simultaneously[2] (the *M:N* communication model).

Even though a videoconference participant could be able to send the video in a high-quality, he or she need not be able to receive the streams from the other ones—for example, when he or she is connected behind network lines providing a sufficient bandwidth for just a single video stream, but which are unable to deliver the amount of data required by the streams from multiple senders simultaneously. There are two basic possibilities, how to cope with such a situation: first, all the streams might be transcoded into lower quality, which would, however, result in wiping all the advantages of the high-quality videoconferences. Thus, the second approach might be more desirable—the current speaker could be determined from a set of active participants, whose video stream could be let as is (or transcoded into a bit lower quality), while the video from the others could be transcoded into a very low quality. All the streams could be further composed into a single video stream (see the example in the Section 6.3), which is delivered to the receivers. This approach reduces the network bandwidth necessary for delivering all the video streams, while still provides the features of the high-quality conferences. Furthermore, this capability might be also useful when a computing power to decode and play many simultaneous high-quality streams in the real-time is not available at receivers' sites.

Figure 10.2: A screenshot of a multipoint H.323 [224] videoconference, where an MCU unit [280] performs the video transcodings and composition.

For both mentioned situations (in fact, for the video transcoding in general), the possibilities of the parallel processing, which are provided by the DiProNN, could be very useful. Inside the node, the received streams could be forwarded to several parallel instances of a single transcoding AP (e.g., on a per client basis), where the down-sampling takes place. If the data packets contain an information, how they should be processed (e.g., as defined by a previous AP), the transcoding might be easily performed depending on client's wishes. After the transcoding, all the transcoded streams might be forwarded to another AP(s), where the synchronization and/or final composition might occur.

[2]Note, that not all the videoconference participants have to necessarily send and receive the video content at a particular time—some of them may just receive the video streams from all the other participants, and both send and receive the audio content only.

Even further, once a DiProNN Program, that provides a composed high-quality video stream, is created, it might be simply concatenated with the DiProNN Program, that provides both the high-quality video stream and a down-sampled video stream(s) (as specified in the previous section). Thus, one can easily choose between the provided streams depending on the (network) situation, that occurs.

10.3 Multimedia Stream Synchronization

Real-time high-quality multimedia transfers are becoming one of the most important applications for current high-speed computer networks. Most of the videos are captured in 2D only, meaning that the reality, which has three dimensions (3D), is reduced to a 2D picture with inevitable loss of some information connected with the depth of the space. To simulate a 3D environment by a stereoscopic image, two streams must be captured (and later displayed in a special way), one for each eye. To provide a natural perception, the quality of the individual streams must be rather high and thus a high-resolution image with a low compression needs to be deployed, which however results in a high data rate. To remove any unwanted effects on the human observer, both streams have to be synchronized when displayed.

When transmitting such a stereoscopic video over the IP networks [78], which are not able to ensure a precise synchronization of the streams, the synchronization must be enforced externally. One possibility is to synchronize and multiplex both the video streams at the source and send them in one data (packet) stream. However, processing of both streams on a single machine might not be feasible depending on a video format used for the transmission—in the cases of an uncompressed HD video or an uncompressed 4K video mentioned before, a non-trivial computing power must be available on the machine to perform such a synchronization.

The second possible approach assumes the synchronization of otherwise independently generated and transmitted video streams at some point in the network, which is close to the display nodes and where a sufficient computing power is available. Such a synchronizing node could be the DiProNN node running two active programs, which communicate with each other via the low-latency control interconnection to ensure the proper streams' synchronization[3]. These APs could be further parallelized, if a higher computing power is necessary (e.g., because of some processing required to be performed on the streams).

10.4 Real-time Computations for Haptic Interactions

The real-time interaction between humans and computers has become an interesting and highly challenging area of research during past years. In particular, the connection between human-computer interaction and computer simulation of biological, chemical, and physical phenomena is focused by many researchers. The goal is to employ the virtual reality to perform operations, which are dangerous or impossible in real world. Besides the visual perception, the *haptic* (touch) sense is successfully employed nowadays to enable the user to touch the virtual objects and scenes in order to get additional information about the simulated objects and processes (such as forces, torques, etc.).

[3]However, to enable the synchronization, the video data must be transferred using an application-level transport protocol, that provides a time-stamping information (for example, the RTP [236]). The synchronization could be then performed in a similar way as described in [78].

Figure 10.3: An example of a linearly polarized stereoscopic front projection located in the Laboratory of Advanced Networking Technologies (Faculty of Informatics, Masaryk University). The bottom picture shows the Paralax setting device (by Apec) and the cameras, which the stereoscopic image is captured with.

A good example is given by a virtual-reality based surgical simulators, which are used for medical training and operation planning. Here, the user interacts with a virtual model of an real object via haptic devices with force feed-back, so he or she can directly touch the virtual object and perform various operations as deforming, tearing, or cutting it. It is clear that the behavior of the simulated object must be realistic, i.e., the behavior of the model must be based on physical laws (e.g., the theory of elasticity). The main issue of an implementation of such a simulator is given by huge computational cost of such a physically-based object simulation and the stable real-time haptic interaction, when the response forces must be updated on a high frequency (over 1 kHz).

Recently, a technique for user interactions with a virtual model of a soft tissue via a haptic device has been proposed in [83, 210]. The approach is based on distributed computations and comprises two main procedures running concurrently. The first one performs massive computations of deformations and forces on a distributed environment composed of independent servers (e.g., clusters or grids), while the second one delivers the computed data to a computer running the real-time haptic and visual loops, where the data are interpolated to obtain the deformations and forces corresponding to the actual position of the haptic device.

There is quite extensive communication performed within the distributed environment: the client distributes the work among the servers, whose computations are driven by the actual position of the haptic device. And vice versa, the servers send the calculated data to the client, which computes the interpolation. Moreover, the servers could communicate among themselves, e.g., to interchange the computed results that can be used as initial estimation for further computations.

The DiProNN could serve as the computational infrastructure for such an application, which prepares the computations of deformations and forces to the clients. Besides the parallel processing, which is highly beneficial for performing such massive computations, the Processing Units could further provide some additional functionality for these applications (so-called *solvers*), like the ability to perform some processing on the GPU accelerators [258]. Last, but not least, the DiProNN's component-based processing can improve the control over the data flow going towards the client; especially, if large models containing a lot of data are computed, then the data can be combined and/or filtered on-the-fly by controlling the proper components (APs) of a particular DiProNN Session.

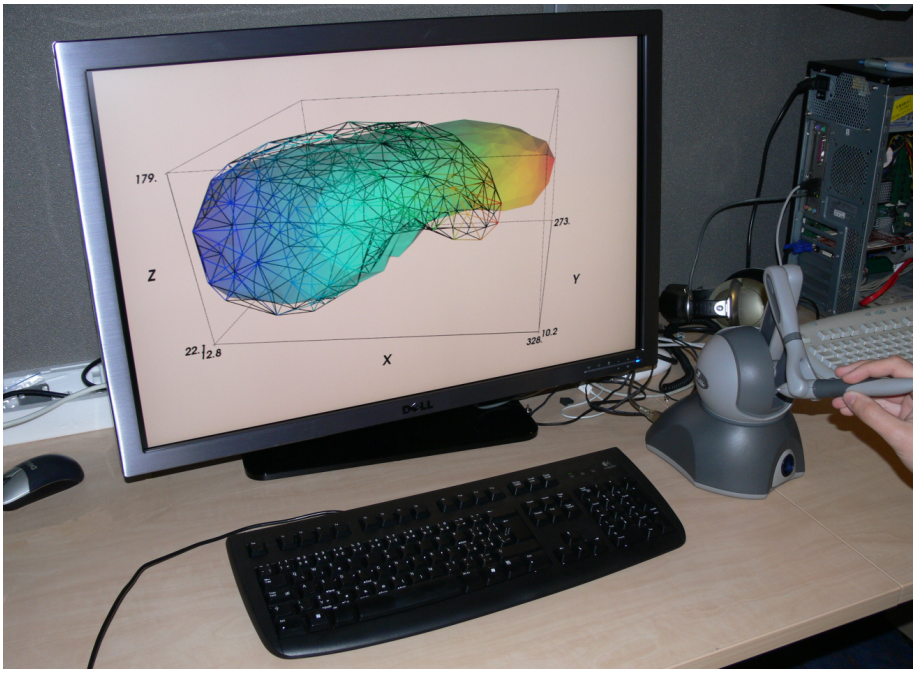


Figure 10.4: Haptic interactions with a deformable model of soft tissue (human liver) using SensAble's PHANToM device with force feedback.

10.5 Data Encryption/Decryption

A communication among network users, which usually passes through a non-reliable networking environment, must often be kept confidential and protected from "prying eyes"—emails, documents, medical records are only a few examples of information that could be required to be secured. However, not only these examples of a non-realtime communication requires securing, but also a real-time information, like video/image data streams during distant medical surgeries, have to be secured as well.

The *cryptography* [250] is a science of scrambling an original message into a code unintelligible to an attacker trying to theft the information[4]. The cryptography is usually based on the use of algorithms [190] to encrypt an original message into a ciphered code as well as to decrypt it on the receiver's side.

Such encryption/decryption functions, that increase the security of passing (real-time) data through a non-trusted network environment, belong to another possible applications of the proposed node. Especially, in cases, when there is a necessity to encrypt/decrypt high-bandwidth data in the real-time (like the mentioned high-resolution medical videos/images)—since most encryption algorithms are fairly complex, an on-the-fly data encryption/decryption requires great computational power, that need not be available on the end hosts. Thus, it could be impossible to do such an encryption/decryption using current commodity PCs in the real-time without additional hardware support.

Even if such a computing power is not available on the end hosts, it might be available on the specialized processing network nodes, like the DiProNN is. Once the computing power, which is provided by a single Processing Unit, becomes not sufficient to cope with an encryption/decryption of such a high-bandwidth data stream in the real-time, the user is able to let the encrypting/decrypting APs to be processed in parallel, which can significantly improve the overall performance.

Nevertheless, since the unsecured information, which passes the links between the sender(s) and the encrypting node as well as between the decrypting node and the receiver(s), is vulnerable to the thefts, the nodes have to be located as close as possible to the sender(s)/receiver(s) (to be accessible via a reliable network) or these links have to be secured in a different way.

10.6 Novel Network Services

As already mentioned before, in current computer networks, the development and deployment of new services is too slow due to best practice and standardization [94]. Thus, to enable faster and more flexible services' deployment, the active/programmable networks have been proposed—the networks, which enable third-parties (end users, operators, service providers, etc.) to inject application-specific services into the network. This allows applications to utilize these services to obtain a required network support—for example, the applications for a multimedia multi-party communication (like reliable multicast) can be deployed and utilized.

[4] In fact, the cryptography not only protects the data from a theft or alteration, but it can also be used for users' authentication, privacy/confidentiality, integrity, and/or non-repudiation [150]. However, these features are not so important for the example described, and thus are not taken into account.

The key challenges for providing such an efficient and scalable reliable multicast service[5] [170] over a wide-area network include: managing bandwidth utilization of bottleneck links, not overloading the sender with retransmission requests, and keeping the latencies of retransmissions as low as possible. At the implementation level, these challenges translate into finding mechanisms for preventing NACK (*Negative ACKnowledgment*) implosion, distributing responsibility for sending transmissions, and limiting the delivery scope of retransmitted packets.

The reliable multicast protocols can take advantages of active/programmable nodes capabilities to combine a reliable multicast processing with the multicast data distribution tree itself. Such nodes may, e.g., check and eliminate duplicate NACKs to prevent overloading of the source, cache limited amount of multicast data to reduce the bandwidth usage and the latency of retransmissions, and/or detect missing data packets and generate retransmission requests to reduce the latency of retransmissions as well (lost packets are detected earlier than the receiver may notice it).

Another example of such novel network services, that could be implemented using the active/programmable nodes, is caching. In traditional networks, there is a need for caches located close to the clients to reduce the network traffic as well as time necessary for information retrieving. Usually, the caches are located near the edge of the network and/or at strategic points in the network. Using active/programmable networks, the caching sites can be decided on-the-fly depending on an actual network traffic. The cache nodes thus can be implemented closer to the clients and can focus on the information required by the clients at a particular time. The network traffic as well as the information's retrieving time thus further decreases.

However, there is one restriction, which slightly limits DiProNN's usage for deploying novel network services in comparison with common active/programmable nodes. That is its unavailability to let the users to use arbitrary pure transport protocols for their applications—as we discuss in the Section 12.1, the node does not include any service information into the packets, so that it has to distinguish among the particular flows according to the information provided by the transport protocols. The users' applications are thus required to use just that pure transport protocols[6], which the node supports.

10.7 Powerful Computing Platform and Distributed Testbeds

Even thought the DiProNN is primarily intended as a data streams' processing platform, it could be also used as a computing and/or testbed platform like the Grids, Clouds, or PlanetLab are. Thus, the users may use provided virtual computers for computing their scientific or technical problems, that require a higher processing power than their personal computer is able to provide. However, in addition to these projects, the DiProNN can further enable the users to upload arbitrary execution environments (e.g., an OS), inside which the computing is performed.

Such a DiProNN Session can consist of one or more virtual machines (possibly without data/control interconnections among them), where each of them contains an application performing the computing. Once the computing finishes, either just the computed results or the whole VM(s) can be automatically/on-demand returned to the user.

[5]Multicast protocols provide a point-to-group communication facility; a multicast protocol is reliable, if it continues to try to deliver an information until it is received by all members of the group.

[6]The application-level transport protocols making use of a supported pure transport protocol are not obviously limited.

Even further, if the session's VM(s) have their DiProNN input port(s) defined and connected to an SSH[7] or a VNC[8] service running inside, the user is enabled to log into the particular virtual machine and control the computing by hand. Thus, a variety of new applications associated with the online computing arises (e.g., distributed testbeds for novel network services).

[7] http://www.openssh.com/
[8] http://www.tightvnc.com/

Chapter 11

DiProNN in Various Virtualization Systems

So far, we have intentionally omitted any mentions about a concrete virtualization system suitable for the DiProNN's implementation. The reason is, that its architecture is general enough in the sense, that it does not rely on a concrete virtualization system. Even further, it does not rely on a specific type of the virtualization systems (the platform-level vs. OS-level vs. process-level VMs, as presented in the Section 2.2)—one thus may choose the proper virtualization system, which mostly fits all the requirements requested from the particular implementation. The DiProNN requires just a few basic principles, which must the employed virtualization system provide—all of them are summarized in the following section.

Nevertheless, even though the DiProNN does not rely on a specific VMs' type, it does not mean, that all its functionalities remain unaffected, once being implemented using any of them—since each type of the virtualization technique can enrich or reduce its functionality, the benefits and drawbacks of employing all the three basic virtualization's approaches are presented in the Sections 11.2, 11.3, and 11.4.

As already mentioned, not all the node's units have to use the virtualization—the Distribution Unit, the Aggregation Unit, and the Control Unit could be implemented using native, non-virtualized systems without any impacts on the DiProNN's functionalities and/or features. The only units, that require to be implemented using a virtualization system, are the Processing Units, and thus the rest of this chapter focuses just on them.

Finally, let us point out, that all the DiProNN nodes[1] in the network do not necessarily have to be implemented using the same virtualization system. Although different nodes will not be able to directly cooperate in such a situation (e.g., to perform inter-node VMs' migrations), this can further enhance the programming flexibility from the user's point of view—since the users' sessions consist of multiple cooperating virtual machines, each of them could be designed for (and thus require) a different virtualization system. Thus, the users might be enabled to employ VMs for multiple virtualization systems within a single session.

[1] Even further, not all the Processing Units of a single DiProNN node have to be implemented using the same virtualization system.

11.1 Required Principles

The Section 2.2 has depicted the general virtualization's architecture consisting of three layers—the hardware layer, the hypervisor's layer, and the virtual machines' layer. The hypervisor's layer provides all the crucial virtualization functions (platform replication, resource management, etc.) necessary for the proper run of the VMs, which operate on top of it. However, all the hypervisor's functionalities as well as all the running VMs have to be controlled somehow. For such reasons, a specialized console or one of the VMs (both denoted as the *Service Domain* within this thesis) usually provide the controlling functionality of the whole virtualized system.

The Service Domain is booted automatically once the hypervisor boots. It usually runs a common operating system and has special management privileges—by the use of hypervisor's services, it is able to control all the other VMs, manage the amount of hardware resources they are enabled to use, as well as has a direct access to the underlying hardware.

In the case of DiProNN, the Service Domain is further required to run a set of modules necessary for the proper functionality of the particular Processing Unit (the VM/AP Management module, the Control module, the Resource Management module, etc.—see the Section 4.2). The modules make use of the Service Domain's services, which is thus required to provide at least the following set of features:

- **VMs' management** – the VM/AP Management module, that runs inside of the Service Domain, has to be able to manage all the VMs running on the particular Processing Unit. For the basic functionality, it requires the Service Domain to provide just the functions for starting and stopping the VMs. However, advanced functions, like VMs' suspending/resuming and/or (live) migration capabilities, are also very beneficial.

  The managed VMs should be provided in an easily distributable form—for example, in the form of a whole "partition" image file or a directory subtree compressed file. The VMs should be also easily duplicable, since this is required to enable the DiProNN's parallel processing (as described in the Section 7.4.1).

- **Packets' forwarding** – the Service Domain has to be further able to "see" all the packets outgoing the VMs (and optionally incoming as well). Such a feature is a fundamental issue for the dynamic forwarding approach, since the Service Domain has to be able to ensure the proper delivery of the packets passing among the VMs (in fact, the APs running inside of them)—as described in the Section 7.3.2, all the packets coming from a certain port of a particular VM have to be forwarded to an appropriate port of a (generally different) VM running the subsequent AP(s), so that the packets might be further processed. Furthermore, if the Service Domain becomes aware of the whole network traffic within a particular Processing Unit, it might detect the intrusion attempts of malicious VMs/APs as well.

- **Resource management** – the Service Domain should be able to allocate the resources requested by the VMs. Moreover, once the resources are distributed among the VMs, the particular virtualization system has to be able to ensure the proper isolation among the VMs, so that they cannot influence each other during the runtime. The set of managed resources should cover at least the basic ones—the CPU time, the memory consumption, the network bandwidth, and the available disk capacity.

The VMs have to be either limited from being able to consume more resources than declared, or guaranteed that the requested resources will be always available, or both. Regarding the resource guarantees, these should be supported by powerful scheduling algorithms satisfying the requirements given in the Section 9.1.1.

Last, but not least, the Service Domain should be also able to monitor a real usage of the resources. This information could be either required by sessions' users, or used for charging the users for real resources' consumptions.

- **DiProNN services inside of the VMs** – besides the described set of the Service Domain's services, the particular virtualization system has to enable to run the APM and PA services inside of the VMs, too. These services should be able to control/communicate with all of the APs running inside of the particular VM as well as with the modules running inside of the Service Domain of the particular Processing Unit.

All these requirements are more or less[2] crucial for an implementation of the node—once a virtualization system does not enable its Service Domain and/or VMs to provide them, the DiProNN cannot be fully implemented using it. In the following sections, we sketch the ways, how these functionalities could be implemented in several existing virtualization systems, that have been described in the Chapter 2.

11.2 DiProNN in Platform-level VMs

The platform-level virtualization systems, which provide an illusion of the whole real machine (computing platform) to the virtual machines running on top of them, are the primary virtualization systems considered for the DiProNN implementation. These systems are assumed during all the descriptions given in this thesis, since they provide the highest flexibility without restricting its features in any way.

As already mentioned during their description, these virtualization systems enable the node to run user-supplied virtual machines with arbitrary execution environments running inside of them. Depending on the virtualization system used, the VMs could be required to be designed for a specific hardware platform (in the case of system-level virtualization systems) or almost no requirements could be requested from them (in the case of whole-system virtualization systems, which are more or less able to emulate an arbitrary hardware architecture).

The platform-level VMs usually implement the Service Domain as a specialized VM (e.g., the Xen) or a service console (e.g., the VMware), both of which usually run a Linux-based operating system[3] used for controlling both the hypervisor and the guest VMs. Thus, the DiProNN modules, which are required for the proper functionality of the Processing Units, can be run as common Unix/Linux applications inside of it.

[2]The only less important requirements are the ones related to the resource management—the resource limitations and resource guarantees. These functionalities are not strictly required for the basic DiProNN functionality—if a resource management functionality is not available in a particular virtualization system, the DiProNN will not "only" be able to reserve/restrict resources requested by the users' sessions.

[3]The virtualization system is usually created upon an existing, previously installed OS, which further serves as the Service Domain. However, some virtualization systems (like the VMware ESX Server) are distributed as a whole—together with a proprietary Service Domain. In spite of this fact, their Service Domains are usually based on the Unix/Linux OS as well—in the case of the VMware ESX Server a customized version of the Red Hat 7.2 distribution has been used—and/or allow user applications to be run inside of them.

Regarding the APM and PA services, which have to run in every VM, these should be provided for various execution environments (OSs) the users might/want to use—if a user wants to use an OS/EE, for which these services are not available, he or she has to implement them (based on the defined API/rules) on his/her own, or ask the node administrator to implement them. Once the services are available, they must be included in the VMs and appropriately set—to be started just after the VM boots and to be aware of all the APs included inside of the particular VM—before being uploaded to the node.

The Service Domains are further responsible for creating a networking infrastructure on every Processing Unit, so that the running VMs are able to communicate with each other and/or host's networking neighborhood. The contemporary virtualization systems usually provide the virtual networking infrastructure based on the routing, bridging, and/or NATing mechanisms. Even thought the DiProNN does not require a concrete mechanism—the employed virtual networking infrastructure could be based on any of these—several requirements must be satisfied:

1. All the Service Domains must be able to communicate (either directly or via ports' forwarding[4]) with all the VMs running on the particular node (even the ones running on different Processing Units),
2. each VM has to be able to communicate with the Service Domain of the particular Processing Unit, which it is running in,
3. if possible, the VMs cannot communicate with each other (to provide higher security guarantees),
4. the Service Domain must be able to "see" and control all the packets being sent from the VMs, that run on the particular Processing Unit—it has to be able to read the source/destination information saved inside of the packets and provide a mechanism to modify packets' destinations, which is required for forwarding the packets among sessions' APs (e.g., using the kernel `iptables` [15], the `ebtables`[5] filters [237], etc.).

11.2.1 DiProNN in Xen

As already mentioned, the Xen is an open-source virtualization system for the x86 processor architecture, whose performance benefits from its paravirtualization technology allowing hosted virtual machines to collaborate with the hypervisor to achieve the best performance.

Even thought the paravirtualization approach requires guest operating systems to be ported (modified) to be able to run on the Xen, the Xen provides the ability to run an unmodified guest OS kernels as well. This ability is enabled by the hardware CPU virtualization, which is provided by the already mentioned Intel VT and AMD Pacifica technologies.

[4]For example, when the NAT mechanism is used, the VMs are not accessible from outside of the host (just the Service Domain is). Thus, in the Service Domain, relevant ports and forwarding rules have to be set during the sessions' establishment, so that relevant data could be delivered to the APs running inside of the host's VMs.

[5]`http://ebtables.sourceforge.net/`

Using the paravirtualization technology, the Xen supports just Unix-like operating systems (Suse Linux[6], Fedora[7], Ubuntu[8], Gentoo[9], etc.) and several specialized execution environments (the Libra OS [12], the LiquidVM OS [300], the JavaGuest OS [138], the Library OS [14], the Maestro-VC [154], etc.). With the hardware assisted virtualization, the Xen further supports unmodified versions of Microsoft Windows[10] and other proprietary operating systems.

The Xen has become the primary virtualization system, which we have used for the DiProNN's prototype implementation, and thus is further discussed and subjected to several performance tests in the following chapter.

Service Domain

The first VM, called *domain 0* or *dom0* in the Xen's terminology, is created automatically during the system boot, and has special management privileges. It builds other VMs/domains (called *domUs*) and manages their virtual devices. It also performs administrative tasks, such as suspending, resuming, and migrating the VMs.

Within the *dom0*, a process called `xend` runs to manage the system. The `xend` is responsible for managing virtual machines and providing access to their consoles. Commands are issued to the `xend` over an HTTP interface or via a command-line tool (`xm`).

Since the *dom0* runs a Linux-like OS, all the Processing Units' modules, that are required to be run in the Service Domain, could be simply run as common Linux applications.

VMs' management

The primary tool for managing the Xen from the console is the `xm` command. It allows all the administrative tasks required by the DiProNN, which are summarized in the following table:

Action	Command
VMs' startup	`xm create <domain>`
VMs' stopping	`xm shutdown <domain>`
VMs' forced stopping	`xm destroy <domain>`
VMs' suspending	`xm suspend <domain>`
VMs' resuming	`xm resume <domain>`
VMs' cold migration	`xm migrate <domain> <dest_host>`
VMs' live migration	`xm migrate --live <domain> <dest_host>`

Table 11.1: The Xen's commands for the VMs' management.

Xen's domains could be represented as physical disk partitions or disk image files, both of which contain the root filesystem of the guest operating system. Since the physical disk partitions are not easy to manage and distribute, the domains' image files are the best option for the DiProNN.

[6] `http://www.novell.com/linux/`
[7] `http://fedoraproject.org/`
[8] `http://www.ubuntu.com/`
[9] `http://www.gentoo.org/`
[10] `http://www.microsoft.com/`

All the domains are described using configuration files located in the `/etc/xen/` directory, which define all the necessary domains' parameters, like the domain name, path to the kernel image, amount of memory and virtual CPUs, network configuration, etc. For further details about the Xen see the users' manual [312].

Virtual networking infrastructure and packet's forwarding mechanism
The Xen supports all the three mentioned virtual network infrastructure mechanisms—the bridging (which is default), the routing, and the NATing [44]. The mechanism, which should be used, could be chosen within the main `xend`'s configuration, located in the file `/etc/xen/xend-config.sxp`.

For a DiProNN implementation, the preferred mechanisms should be the bridging or routing, both of which allow all the VMs (domains) to appear on the network as individual hosts[11]. Thus, all the VMs are directly accessible from outside of the particular unit (especially from all the other Service Domains), while all the packets could be appropriately filtered and forwarded by the `iptables` or `ebtables` rules defined in the Service Domain [238].

Even thought the bridging and routing mechanisms are the preferred ones, there is at least one situation, when the NATing mechanism could be more suitable—that is the case of the minimal DiProNN's architecture (the DiProNN node consisting just from a single Processing Unit), when the NATing avoids interferences with the external network.

APM/PA services inside of the VMs
Both the APM and PA services should be implemented for all the OSs/EEs being able to run in the Xen's domains. The communication between the services and the active programs running inside of the particular VM could be performed by any form of an interprocess communication, like (network) sockets, named pipes, message queues, etc. [105, 251]. The communication between the services and the VM/AP Management and Control modules, which they have to communicate with, could be performed using a standard network communication (the modules have to be available on a Service Domain's well-known port).

Resource management
The Xen's support to resource limitations, guarantees, and monitoring is quite poor. Outside of the box, the Xen is able to limit the amount of memory, which a particular domain is allowed to consume (the `memory` parameter in the domain's config file or dynamically via the `xm` command) and limit the amount of the CPU time the domain is enabled to consume (dynamically via the `xm` command depending on the CPU scheduler used [61, 226]). The VMs' networking bandwidths could be enforced externally by the shaping functionality of the *Traffic Control (`tc`)* tool [42] (available under the Linux OS) in the particular Service Domain, while a simple resource usage monitoring can be performed using the Xen's `xentop` tool.

Nevertheless, there are some attempts to enrich the Xen's resource management system. For example, in [109], the authors propose the design of a set of primitives addressing the problem of the Xen's performance isolation—the proper accounting technique for CPU shares considering the CPU times consumed by domain's device drivers (running in their isolated driver domains and thus not directly included in the domain's resource consumptions). The same authors further propose a novel performance monitoring tool,

[11] Remember, that the node operates on a private network segment, as mentioned in the Chapter 4.

called *xenmon* [110], which reports a detailed resources' usage of the VMs, and which thus provides an additional insight into the shared resource access and resource scheduling.

11.2.2 DiProNN in VMware

Regarding the VMware products, the DiProNN could be implemented by more of them. The most complex virtualization system, which is provided by the VMware, the VMware ESX Server, is a part of the non-free *VMware Infrastructure* product, and provides fine-grained, policy-driven resource allocation mechanisms, resource optimizations, high-availability assurances, VMs' migrations, etc. Even thought the VMware Infrastructure further contains additional services besides the VMware ESX Server (e.g., the VirtualCenter Server, the VMware Infrastructure Client, the VMware High Availability, etc. [308]), the VMware ESX Server could be used independently on the others, and thus could serve as a basis for the DiProNN implementation.

The other VMware product, the VMware Server, is a free x86 and x86-64 virtualization product, which serves as a standalone platform for running multiple virtual machines on the same host. Even though the VMware Server is mainly dedicated to end users (focusing on a GUI-based interface), by using the `vmrun` utility it could be controlled from the command line as well, and thus becomes usable for the DiProNN as well. However, it does not provide as many features as the VMware ESX Server does—e.g., it is not able to perform VMs' live migrations in any way (although the cold ones are still available).

Similarly to the VMware Server, the VMware Workstation is another VMware's non-free virtualization product focusing mainly on the desktop user applications.

All the VMware products support a large variety of guest operating systems, including both the 32-bit and 64-bit versions of: Windows Server, Windows Vista, Windows XP, Mandriva Linux[12], Red Hat Linux[13], Suse Linux, Ubuntu Linux, etc. [307, 310, 311], and most of the specialized execution environments depicted in the Xen's section.

Service Domain/Service Console

The VMware ESX Server is currently available in two versions—the *ESX Server* version, which includes a built-in Red-Hat Linux-based service console, and the *ESXi Server* version, which does not. Recently, the VMware ESXi Server has been made available for free (however, the ESX version still remains non-free).

Similarly to the ESXi Server, the VMware Server and the VMware Workstation products are distributed without their own service console. Thus, all of these could be installed on any Linux-based system[14], which then serves as the service console, and which is able to run all the DiProNN modules necessary.

VMs' management

The primary tool for managing the VMware ESX Server is the `vmware-cmd` utility, which performs various operations on the virtual machines, including registering a virtual machine (on the local server), setting/getting the power state of a virtual machine, etc. The basic set of its functions, that are required by the DiProNN, is depicted in the following table:

[12] `http://www2.mandriva.com/`

[13] `http://www.redhat.com/`

[14] These product could be installed on a Windows-based host as well, however, for the DiProNN, the Linux-based host is more suitable.

Action	Command
VMs' startup	`vmware-cmd <vm-cfg-path> start soft|trysoft|hard`
VMs' stopping	`vmware-cmd <vm-cfg-path> stop soft|trysoft`
VMs' forced stopping	`vmware-cmd <vm-cfg-path> stop hard`
VMs' suspending	`vmware-cmd <vm-cfg-path> suspend soft|trysoft|hard`
VMs' resuming	`vmware-cmd <vm-cfg-path> start soft|trysoft|hard`
VMs' cold migration	*suspend & resume*
VMs' live migration	directly unavailable (by default ensured by the VirtualCenter Server—another product belonging to the VMware Infrastructure), but available through the VIMSH scripting [306]

Table 11.2: The VMware ESX Server's commands for the VMs' management.

Regarding the VMware Server and the VMware Workstation products, these are controlled by the `vmrun` utility. The VMs' management functions satisfying the basic DiProNN requirements are summarized in the Table 11.3.

Action	Command
VMs' startup	`vmrun -T server -h hostnameOrIP start <vm-cfg-path> nogui` `vmrun -T ws start <vm-cfg-path> nogui`
VMs' stopping	`vmrun -T server -h hostnameOrIP stop <vm-cfg-path> soft` `vmrun -T ws stop <vm-cfg-path> soft`
VMs' forced stopping	`vmrun -T server -h hostnameOrIP stop <vm-cfg-path> hard` `vmrun -T ws stop <vm-cfg-path> hard`
VMs' suspending	`vmrun -T server -h hst_IP suspend <vm-cfg-path> soft|hard` `vmrun -T ws suspend <vm-cfg-path> soft|hard`
VMs' resuming	common start of a previously suspended VM
VMs' cold migration	*suspend* & *resume*
VMs' live migration	*not supported*

Table 11.3: The VMware Server's and VMware Workstation's commands for the VMs' management.

Every VMware's domain is described by a VMX file, which contains all its configuration parameters. Similarly to the Xen, all the VMware products are able to run the domains represented as physical disk partitions or disk image files, and thus similar facts apply as for the Xen virtualization system.

Virtual networking infrastructure and packet's forwarding mechanism
For all the mentioned products, the VMware provides bridged, network address translation (NAT), and host-only networking options to configure the virtual networking. The bridged and NAT networking mechanisms are more or less the same as in the case of the Xen, while the host-only networking mechanism creates a network, that is completely contained within the host computer—it provides just a network connection between the virtual machine and the host computer (the service console). [307, 310, 311]

Similarly to the Xen, the best option for the DiProNN's implementation is the bridged virtual network, which allows all the guest VMs to be directly accessible from outside of the particular Processing Unit. The usage of the kernel's `iptables` or the `ebtables` filters to ensure proper packets' forwarding is also possible within the VMware's service console.

APM/PA services inside of the VMs
The same facts apply as for the Xen virtualization system (see the previous section).

Resource management
The VMware ESX Server excels in this subject—there are many resources [302], that could be limited, allocated, and/or monitored during the VMs' runtime. In the VMware ESX Server, the main tool for managing the CPU, memory, and disk resources is the `vmware-cmd` utility, which—by setting defined variables [309]—allows their limitations and/or allocations. Even thought the variables could be read as well, there is more comfortable tool for performing the resources' usage monitoring, represented by the `esxtop` utility.

Against the VMware ESX Server, neither the VMware Server nor the VMware Workstation provide capabilities for the resource management. Thus, if the resources could not be managed/monitored externally from the Service Domain (the host OS), there is no way to provide the resource management by the products themselves.

One could notice, that we have not mentioned the network traffic management for the VMware ESX Server. The reason is, that there is no way to manage the network resources throught the `vmware-cmd` command, since there are no variables allowing to define the allocations and/or limitations for it [309][15]. However, the network resources in the VMware ESX Server as well as in the VMware Server and the VMware Workstation products could be managed in the same way as in the Xen virtualization system—by the Traffic Control utility (`tc`) from the host OS.

11.2.3 DiProNN in QEMU

As already depicted, the QEMU operates in two modes: in the *full-system mode* the QEMU emulates a full system (for example a PC) including one or more processors and various peripherals, whilst in the *user-mode emulation mode* it launches just processes compiled for one CPU architecture on another one. Even thought the node can be based on the user-mode emulation only (see the Section 11.4 describing DiProNN's implementation in process-level VMs), in this section we assume just the QEMU's full-system mode, which allows more features and does not restrict the node's functionalities as compared to the user-mode.

Nevertheless, the user-mode emulation could be combined with the full-system mode as well—in such a case, the user-mode emulation can serve for the standalone APs, whilst the full-system mode for the whole VMs. On the one hand, this can provide lower overhead necessary for running the standalone APs, while on the other, these APs have to contain the APM and PA services implemented on their own, similarly as in the case of process-level virtualization systems.

Since the QEMU is able to emulate many hardware targets (the comprehensive list can be found in [177]), it represents a very versatile virtualization system enabling the highest execution environments' flexibility currently available.

Service Domain
No matter which mode the QEMU runs, each QEMU's VM contains its own console—called the *QEMU Monitor*—through which the particular VM is controlled. By default, the monitor is accessed from within the guest OS by holding down the *CTRL-ALT-2* key

[15] There are just variables providing statistical information about amounts of data received/transmitted, amounts of packet received/transmitted, etc. [309]

combination (and the *CTRL-ALT-1* key combination to switch back to the guest OS). However, this approach is not suitable for the DiProNN, since the VMs have to be controlled externally from the host OS—the monitor has to be exposed to the host.

Thus, when starting a VM, the QEMU console has to be redirected[16] to a network socket, which could be later accessed from the host's OS by various applications, like the `telnet` or the `netcat` Linux utilities.

Regarding the host OS, the QEMU can be installed on an existing Linux, Windows, or MacOS X operating systems. Since the QEMU versions for the Windows and MacOS X operating systems are just experimental, the QEMU for the Linux OS is the best choice for a particular implementation. The host's OS then serves as the Service Domain, which throught the network sockets controls all the VMs, and further runs all the necessary Processing Unit's modules.

VMs' management

As mentioned before, the QEMU Monitor serves as the main management and monitoring tool in the QEMU. Using various commands the monitor allows to inspect a particular running guest OS, change its removable media and USB devices, take screenshots and audio grabs, and control various aspects of the particular virtual machine. The virtual machines are represented as image files containing the whole guest OS.

Once a virtual machine is started and its QEMU Monitor becomes accessible from the host's OS, it can be further controlled through the following set of commands:

Action	Command
VMs' startup	`qemu -monitor telnet::[port],server,nowait [disk_img]`
VMs' stopping	`system_powerdown` command via the `netcat` or `telnet`
VMs' forced stopping	`quit` command via the `netcat` or `telnet`
	or `kill -9 <QEMU_VM_pid` command from the host OS
VMs' suspending	`stop` command via the `netcat` or `telnet`
VMs' resuming	`cont` command via the `netcat` or `telnet`
VMs' cold migration	*suspend the VM*
	`migrate file://<file_location>` command via the `netcat` or `telnet`
	`qemu -incoming file://<file_location>` on the destionation
	resume the VM
VMs' live migration	same as the cold one, but without *suspending* and *resuming* the VM

Table 11.4: The QEMU's commands for the VMs' management.

Note, that for starting the VM, the `qemu` command serves just for the PC guests. For non-PC guests, the QEMU has to be started using different commands, like the `qemu-system-ppc` command for the PowerPC architecture, the `qemu-system-sparc` command for the Sparc32 system architecture, the `qemu-system-arm` command to simulate the ARM machine, etc.—see [177].

Furthermore, the QEMU does not directly support the VM's suspending. The reason is, that if a whole VM has been suspended, it would has never been resumed, since the suspending would stop its QEMU Monitor as well. Thus, for "suspending" a VM, the QEMU emulation could be stopped only, and started (resumed) again later.

[16]The redirection could be performed by passing the `-monitor telnet::[port],server,nowait` option to the QEMU when starting the VM.

Last, but not least, the QEMU migration process assumes, that the migrated VM's image is accessible on both source and destination hosts (located on a shared storage, e.g. using NFS [244], AFS [52], or DFS [252] services).

Virtual networking infrastructure and packet's forwarding mechanism
By default, the QEMU uses a completely user-mode network stack that bridges to the host's network. Even thought it acts as a firewall and does not permit any incoming traffic, the appropriate host's ports could be redirected by the Service Domain to the guest. However, this user mode network stack is not suitable for a particular DiProNN implementation, since it does not support protocols other than the TCP and the UDP.

To provide full networking capabilities for a guest OS, the TAP interfaces[17] must be used. Once a TAP interface is created for each QEMU virtual machine before its startup, the VM creates its own virtual interface and connects it to the created TAP interface (via the `-net nic -net tap,ifname=tap0` options). Then, the traffic flows from and to the VM could be ensured and/or filtered by the host's `iptables` mechanism similarly as in the other virtualization systems (or, when the interfaces are connected to a bridge, by the `ebtables` mechanism as well).

APM/PA services inside of the VMs
The same facts apply as for the Xen virtualization system (see the Section 11.2.1).

Resource management
When starting a VM, the QEMU is able to limit the amount of memory, which the VM is able to consume (via the `-m` option), and which can be further checked by the `info mem` command and dynamically altered by the `balloon <valueMB>` command, both of them sent to the VM's monitor during its runtime.

Since the running VMs appear as independent processes inside of the host OS, the other resource limits/guarantees/monitoring depend on the capabilities of the particular used OS (e.g., CPU limits via the `cpulimit` utility[18], disk bandwidth via the `ionice` utility[19], etc.).

Regarding the network bandwidth limitations/guarantees, these could be provided by the `tc` utility similarly as in the case of both the previously described virtualization systems.

11.3 DiProNN in OS-level VMs

As mentioned in the Chapter 2, the OS-level virtualization systems provide just an "multiplication" of an existing system (usually a Unix/Linux-based OS). Although they do not enable their users to run an arbitrary operating system in the VMs, they usually provide better performance than the platform-level VMs—these might become useful for a particular node implementation not requiring arbitrary OSs to be supported.

In these virtualization systems, the Service Domain is represented by the core OS running on the host system. The host's OS contains an OS kernel, which provides a kernel service abstraction layer ensuring the isolation and security of resources among different VMs (so-called Virtual Environments or Virtual Private Servers). This abstraction service

[17] `http://vtun.sourceforge.net/`
[18] `http://cpulimit.sourceforge.net/`
[19] `http://linux.die.net/man/1/ionice`

then makes the VEs, which use a relevant part of the host's OS kernel, to feel and behave like standalone servers.

The usage of a single kernel for all the VEs consequences in all the benefits and drawbacks these systems provide. Besides the lower flexibility, which results from the impossibility to run an arbitrary OS (both the guest and host OSs must be the same), the usage of a single kernel further consequences in lower isolation and security capabilities, as compared to the platform-level virtualization systems—the host OS's kernel exposes the same bugs and potential security holes to all the VMs, which can compromise each other throught them. However, this drawbacks are compensated by a better performance and a very low overhead, which enables to maximize the usage of underlying system resources.

Moreover, the OS-level virtualization systems usually provide finer-grained resource management capabilities than the platform-level ones. This is allowed by the fact, that every VE is represented by a set of processes, so that the OS's kernel (which usually provides some resource management capabilities for the processes) is able to manage the VEs in a similar way as the native processes.

Concerning the DiProNN's implementation, the host OS can run all the modules, which are required for the proper functionality of the Processing Units, as native OS's applications. Even though all the VEs share a single kernel, the APM and PA services need not necessarily be the same for all of them—the VEs can run different distributions/libraries on top of the unique kernel, and thus might require a slightly different versions of the services. However, the number of different versions, that should be provided, is much lower than in the case of the platform-level VMs.

Regarding the networking infrastructure and its features, the same requirements must be satisfied as for the platform-level virtualization systems.

11.3.1 DiProNN in Linux-VServer

The Linux-VServer technology is a soft partitioning concept based on *Security Contexts*, which allows creation of many independent *Virtual Private Servers (VPS)* (one VPS per security context), that run simultaneously on a single physical server at full speed, efficiently sharing host's hardware resources. It is able to run on various HW platforms, e.g., the ARM, IA64, x86, x86_64, etc. [298]

The Linux-VServer is based on the Linux OS's kernel, which is enhanced by OS-level virtualization capabilities. As already mentioned, the virtual servers (the guests) share the same kernel—they do not run a kernel on their own (like, e.g., the Xen's or VMware's guests do), which requires them to be based on the Linux OS as well. In spite of this fact, several Linux distributions are supported by the Linux-VServer—for example, the CentOS, the Debian, the Fedora, the Gentoo, the Ubuntu, etc.

Service Domain

Once a Linux-based operating system is installed on a machine, the Linux-VServer is installed as a patch of an appropriate Linux vanilla kernel[20] together with the *util-vserver* utility dedicated to its management. The original OS thus becomes Linux-VServer's Service Domain—all the modules required for the proper Processing Units' functionality can be thus run as common Linux applications inside of it.

[20]The "vanilla" kernel is an unmodified version of the Linux kernel, released by Linus Torvalds. Each distribution then takes these vanilla kernels and adds their own type of flavoring (e.g, bug fixes, functionality enhancements, etc.)

VMs' management
In the Linux-VServer, the main VMs' management tool is the *util-vserver* utility, which, besides the others[21], includes the `vserver` command. Even thought the command provides several functions, just starting and stopping the VPSs are meaningful for the DiProNN (see the Table 11.5).

Action	Command
VMs' startup	`vserver <vserver_name> start`
VMs' stopping	`vserver <vserver_name> stop`
VMs' forced stopping	*not supported*
VMs' suspending	*not supported*
VMs' resuming	*not supported*
VMs' cold migration	*not supported*
VMs' live migration	*not supported*

Table 11.5: The Linux VServer's commands for the VMs' management.

Each VPS (VM) is represented by a standard Linux root directory tree, which is located in a directory of the host OS (the default location is `/vservers/<vserverID>/`). This makes the guest systems easy to distribute, e.g., via a compressed file. Finally, the configuration files for the VServer guests are located in the `/etc/vservers/<vserverID>/` directory.

Virtual networking infrastructure and packet's forwarding mechanism
The Linux-VServer's networking is based on an isolation rather than virtualization, so there is no additional overhead for the passing packets. The isolation is ensured by the IP aliases [217] mechanism—the host creates several aliases for the particular network interface(s), throught which the guests are connected to an external network. To ensure the proper data flows in DiProNN, the `iptables` mechanism can be used within the Linux-VServer.

However, since the IP aliases do not allow setting different MAC addresses for the aliased interfaces, the VPSs' IPs must be assigned statically (the dynamic address assignments, like the DHCP offers, are not straightforwardly usable).

APM/PA services inside of the VMs
Similarly to the most OS-level virtualization systems, the Linux-VServer allows starting the commands, which belong to and will run in a particular VPS, from the host OS (from the Service Domain). Thus, one may consider the APM service useless for these systems, since all the required APs could be started externally. However, even thought the APs could be started in such a way, the APM service also performs another functionalities, which cannot be ensured externally—e.g., the communication with the distribution APs to control the number of parallel instances of an parallelizable AP. Moreover, the independent APM service is much clearer solution than the external APs' management.

The other requirements as well as services' communication models are very similar to the ones described for the platform-level virtualization systems.

[21] The other commands are the `vserver-info` command, which gives info about the Linux-VServer program itself, the `vtop` command, which shows the top of all VPSs, the `vpstree` command, which provides a tree `ps` view of processes of all VPSs, and the `vps` command, which shows processes of all VPSs.

Resource management
The standard Linux kernel is able to limit various system resources per process. The Linux-VServer has extended this per process system, so that it is able to provide resource limits for the whole VPSs, not just single processes. Additionally, several new limits, that are missing in the standard vanilla kernels, have been introduced.

For example, the Linux-VServer is able to limit the CPU time dedicated to a VPS, the maximum file size and the maximum number of processes which a VPS is allowed to create, the maximum number of opened sockets, etc.[22] The resource limits and monitoring is performed via the *procfs* [199] filesystem, the traffic management could be provided by the `tc` tool, and the disk space limits could be performed by the `vdlimit` utility[23].

11.3.2 DiProNN in OpenVZ

The OpenVZ is another container-based virtualization system for the Linux OS. It is able to create multiple secure, isolated *containers* (similar to VPSs in the Linux-VServer) on a single physical server enabling better server utilization and ensuring that the applications do not conflict.

Similarly to the Linux-VServer, the OpenVZ is also a modified Linux kernel, which is enriched by various functionalities. It could be installed on the Fedora Core 3 or 4, the Red Hat Enterprise Linux 4, or the CentOS 3.4 or 4 distributions[24], which are configured in a certain way (creating a partition for the OpenVZ, modifying the boot loader, etc.). The OpenVZ is able to run various Linux-based distributions inside of the containers, e.g., the CentOS, the Debian, the Fedora, or the Ubuntu distributions.

Service Domain
Similarly to the Linux-VServer, the OpenVZ's Service Domain is the Linux-based host OS it is installed in.

VMs' management
The `vzctl`, the `vzquota`, and the `vzpkg` are the basic utilities used for controlling the OpenVZ. The `vzctl` is dedicated to perform administrative tasks on the containers (create, destroy, start, stop, etc.), the `vzquota` controls the containers' quotas, and the `vzpkg` is used to work with containers' templates. Besides these, the `vzdump` utility can be further used to make consistent snapshots of running containers and the `vzmigrate` utility for containers' migrations.

Similarly to the Linux-VServer, the containers are represented by a standard Linux root directory tree, which is located in the `/vz/root/<containerID>` directory by default. The containers' systems thus could be easily distributed in the form of compressed files as well. For the sake of completeness, the guests configuration files are located in the `/etc/vz/conf/<containerID>.conf` files.

Virtual networking infrastructure and packet's forwarding mechanism
The OpenVZ provides two mechanisms for networking the guests—the *venet (Virtual*

[22] The detailed list of resources, which the Linux-VServer is able to control, can be found at `http://linux-vserver.org/Resource_Limits`

[23] `http://linux-vserver.org/Disk_Limits_and_Quota`

[24] Even though the OpenVZ is supported just for the mentioned distributions, its installation on the Debian Etch and Lenny distributions is also possible—see `http://wiki.openvz.org/Installation_on_Debian`

Action	Command
VMs' startup	`vzctl start <containerID>`
VMs' stopping	`vzctl stop <containerID>`
VMs' forced stopping	`vzctl exec <containerID> kill -9 -1`
VMs' suspending	`vzdump --suspend <containerID>`
VMs' resuming	`vzdump --restore <vz_dumpfile> <containerID>`
VMs' cold migration	`vzmigrate <dest_host> <containerID>`
VMs' live migration	`vzmigrate --online <dest_host> <containerID>`

Table 11.6: The OpenVZ's commands for the VMs' management.

NETwork) mechanism, which is more or less similar to the networking mechanism provided by the Linux-VServer, and a bit newer *veth (Virtual eTHernet)* mechanism, which allows more features[25].

The former mechanism (the *venet*) behaves like a point-to-point connection between a container and the host system. It is a bit faster and more efficient than the *veth* mechanism, however, just the OpenVZ host node administrator can assign the IPs to the containers.

Against it, the latter mechanism (the *veth*) provides Ethernet-like devices, which can be used inside of the containers. These devices are fully virtualized—they have their own MAC address, therefore, once the virtual devices are bridged to an external device, the containers are able to request their IP address, e.g., via DHCP requests.

Regarding the packets' forwarding mechanism, the *venet*-based networking allows the usage of the `iptables` mechanism only, while for the *veth*-based networking one may choose between the `iptables` and `ebtables` mechanisms.

APM/PA services inside of the VMs
The same facts apply as for the Linux-VServer (see the Section 11.3.1).

Resource management
In comparison with the other OS-level virtualization systems, the OpenVZ does a very good job in this area—over 20 crucial resources can be set live, while the VM is running, or optionally saved to be re-initialized to new values after a reboot [254]. The parameters, which are able to influence both the containers' limits and guarantees, could be set globally in the OpenVZ's main configuration file, separately for each container inside of its own configuration file, or live from the command line using the `vzctl` utility. Regarding the network traffic management, again, the bandwidth limits could be enforced externally by the *Traffic Control (`tc`)* tool [42] in the OpenVZ.

To monitor resources' consumptions, the statistics (including the current usage, the maximum usage, as well as the number of unsuccessful attempts to allocate a particular resource) can be obtained from the `/proc/user_beancounters` file. Furthermore, a number of the memory-related parameters can be monitored by the `vzmemcheck` utility.

[25] `http://wiki.openvz.org/Differences_between_venet_and_veth`

11.3.3 DiProNN in FreeBSD Jails

An extension of the traditional `chroot` environment in the FreeBSD OS is the mechanism of *Jails*. A jail enables the creation of various different virtual machines, each of them having their own set of utilities installed and their own configuration. Thus, the processes running inside of them are isolated and cannot interfere with each other.

Similarly to the other OS-level virtualization technologies, the FreeBSD Jails do not achieve the true virtualization because they do not allow the virtual machines to run different kernel versions than that one of the base system. Thus, all the VMs are required to run just the FreeBSD kernel-compatible OSs—in fact, the same FreeBSD OS (even thought different libraries are possible). However, Linux-based distributions ported to support the FreeBSD kernel are also possible (e.g., the Debian GNU/kFreeBSD[26] and Gentoo/Alt[27] projects).

Service Domain
Similarly to both OS-level virtualization systems described before, the Jails' Service Domain is represented by the FreeBSD host OS.

VMs' management
Even thought the jails are also able to run a single process only, their ability to behave like a VM is more important for the DiProNN. Such a jail is bound to a particular file system's root having a similar structure as the host OS's one. Each jail is bound to a single IP address and (in the default scenario) competes for the host's HW resources with the host OS and the other jails.

The jails are controlled by the `jail` command, which allows several functions related to their creation and management. The functions, which are important for a DiProNN implementation, are summarized in the following table:

Action	Command
VMs' startup	`jail <jailrootdir> <jailhstname> <jailIP> /bin/sh /etc/rc`
VMs' stopping	`jexec -U root <jailID> /etc/rc.shutdown`
VMs' forced stopping	`jexec -U root <jailID> kill -KILL -1`
VMs' suspending	*not supported*
VMs' resuming	*not supported*
VMs' cold migration	*not supported*
VMs' live migration	*not supported*

Table 11.7: The FreeBSD Jails' commands for the VMs' management.

Similarly to both the previous OS-level virtualization systems, the FreeBSD Jails are represented by the FreeBSD root directory tree located in the `/usr/jail/<jailID>` or `/data/jail/<jailID>` directories. The jails' configuration files are located in the `/etc/sysconfig/jail/<jailID>` files.

Virtual networking infrastructure and packet's forwarding mechanism
Similarly to the Linux-VServer, the FreeBSD Jails' networking is based on the IP aliases. Thus, similar facts apply as the ones described in the Section 11.3.1.

[26] `http://www.debian.org/ports/kfreebsd-gnu/`
[27] `http://www.gentoo.org/proj/en/gentoo-alt/`

However, since the `iptables` mechanism is unavailable in the FreeBSD, the packets' forwarding mechanism should be ensured, e.g., by the FreeBSD's *Network Address Translation daemon*, commonly known as the `natd`.

APM/PA services inside of the VMs
The same facts apply as for the Linux-VServer (see the Section 11.3.1).

Resource management
The FreeBSD Jails are able to limit the CPU time and the memory usage of the jails running on the particular system (via the `sysctl` system utility). The CPU time limitations are implemented by providing each jail with a number of CPU shares and tracking the estimated CPU usage of the tasks, that run in that jail. If the ratio of the jail's estimated CPU usage to the total CPU usage exceeds the ratio of the jail's CPU usage shares to the total CPU usage shares outstanding, the jailed processes have their priorities decreased until the ratio of actual usage (estimated CPU) drops below permitted usage (shares). In short, more shares a jail has, more often its processes will run. Unjailed processes do not subject to this regime.[28]

Regarding the memory limitations, a kernel thread periodically traverses all the processes in the particular jail and sums the amount of memory being consumed by them. If a memory limitation is set and the jail's memory exceeds the pre-set limit, the thread asks the virtual memory system to reclaim some of the memory being used by the jail's processes.

11.4 DiProNN in Process-level VMs

The process-level VMs, which provide virtual environments just for particular processes, do not belong to the primary virtualization systems suitable for DiProNN's implementations. They limit most of the presented features, although on the other hand, they slightly beat the other virtualization systems with the ease of their application. Nevertheless, the possibility of implementing the node using these systems is mentioned especially because of interestingness purposes (as a proof for a complete independence on the virtualization systems) rather than because of its importance.

For the proper functionality, these systems require a single host operating system, above which all the virtual machines (each of them consisting of a virtual environment and a single active program) run as common OS's applications. Since most of the process-level virtualization systems are implemented for several common OSs, no specialized OS is necessary. This makes their application a bit easier—once a suitable OS is installed, the virtualization system is installed into it as a common application without any needs to modify the OS's kernel.

The OS then behaves as the "Service Domain", even though it is not the Service Domain in the right sense. In both the virtualization systems' types described before, the Service Domain is a part of the virtualization system, both of which run all the time the node is started. Against it, the process-level virtualization systems do not have their own Service Domain at all—they are externally controlled (started/stopped) together with a particular process, which they provide a virtual environment to, from the OS. Thus, no virtualization system is started, if there is no active program running.

[28] `http://wiki.freebsd.org/JailResourceLimits`

As already depicted before, these virtualization systems provide the lowest flexibility from the ones mentioned, since they enable the users to upload just standalone active programs into the DiProNN. These active programs are, as sketched in the previous paragraph, further provided by their own and complete virtual environment. From the OS's view, all the virtual environments (together with the APs running inside of them) then behave as native OS's applications.

The security and isolation capabilities are also slightly limited, even though still higher than in the case of native OS's applications. The reason is, that all the virtual environments share the same OS's kernel, through which a malicious application can compromise the other ones. However, these attacks have to be performed through the virtualization system, which could detect and deny them.

Similarly to the OS-level virtualization systems, one may also consider the APM service useless for the process-level virtualization systems—the virtual environments contain just a single active program, which is started and further controlled by the host OS, and thus the APM has nothing to control. However, similarly to the OS-level VMs, the contrary is the case—even thought the APM is useless for common APs, it is essential for the distribution active programs. Nevertheless, the implementation of both the VMs' services (APM and PA) differs from the previous cases—since the virtual environments are able to run just a single AP, both the APM service and the PA service thus have to be implemented as AP's functions/procedures, which communicate with the appropriate module(s) running in the Service Domain (the host OS).

Since the process-level VMs behave as common processes from the host OS's point of view, they are not provided by their own network stack. Rather, they use the host OS's network stack and can be distinguished just by the network ports they use. Thus, the forwarding mechanism among the VMs is performed by forwarding the packets to the *Service Domain's IP address* and to the relevant VM's port, that a particular communication interface is associated with.

Last, but not least, the resource management system of the node based on a process-level virtualization system depends on the host OS's capabilities. Since these virtualization systems could be usually run upon many OSs, one may use a highly specialized OS/kernel providing complex resource management functions, e.g., the Rialto OS [136], the Nemesis OS [172], the QLinux OS [253], the RT-MACH OS [263], the Maruti OS [174], etc. Thus, fairly complex resource limitations/guarantees could be provided.

Since the implementations based on different process-level virtualization systems differ just in the execution environments they provide, we do not describe them independently as in the cases of platform-level and OS-level ones.

Service Domain
The Service Domain is obviously represented by the host OS, that the VMs run in. Thus, all the necessary Processing Unit's modules can be run as native applications inside of it.

VMs' management
Since the process-level VMs behave as native host OS's applications, the VMs' management capabilities depend on the capabilities of the particular host OS. Besides the common functions, like starting and stopping the applications, these could include more advanced functions, like suspending, resuming, and/or migrating the applications as well (either as native OS functions or performed by third-party utilities).

For example, in the Linux OS, which is more or less the preferred host OS for the DiProNN, the Java programs are started via the `java <application>` command, the .NET programs via the `ilrun <application>` command, etc. Stopping the programs could be performed by the *SIGTERM* signal sent to the application (`kill -SIGTERM <ProcessID>`), while a forced stopping can be achieved by the *SIGKILL* signal (`kill -SIGKILL <ProcessID>`) [35].

Suspending and resuming the processes could be performed by the *SIGSTOP* (pause) and *SIGCONT* (resume) signals. Such a suspend, however, does not keep the suspended process's state between system restarts, and thus much safer suspends could be performed by third-party checkpointing utilities[29] (for example, by the *CryoPID*[30] or the *Dynamite checkpointer*[31]), which are able to suspend the process's state to a file. Such utilities can further perform process migrations between physical hosts as well.

Virtual networking infrastructure and packet's forwarding mechanism
There is no need to perform a virtual networking infrastructure for the VMs, since they behave as native host OS's applications. Thus, just the host OS's network stack, which is later used by all the VMs, has to be set properly.

Regarding the forwarding mechanism, this depends on the capabilities of the particular host OS as well—for example, in the Linux OS, the mentioned `iptables` or `ebtables` mechanisms could be used.

APM/PA services inside of the VMs
Both the services have to be implemented by the APs themselves—they could be provided as functions, procedures, or whole classes performing required functionality.

Resource management
As already mentioned in the preamble of this section, the resource management and monitoring features depend on the host OS's capabilities. Thus, since the VMs behave as native applications from the host OS's point of view, the resource management/monitoring can be provided for individual processes (APs) in a common way.

[29] `http://www.checkpointing.org/`
[30] `http://cryopid.berlios.de/`
[31] `http://www.science.uva.nl/research/scs/Software/ckpt/`

Chapter 12

Evaluation

In general, there are two methods, that could be used for an evaluation of research contributions such as those presented within this thesis—the qualitative and quantitative evaluation. Since the presented contributions cover mainly the architectural design of the proposed architecture, the main focus of the evaluation, which is depicted in this chapter, lies in the qualitative aspects.

Nevertheless, the initial goal of our work is to illustrate the benefits of employing the virtualization in the active/programmable networks area. Besides evaluating all the benefits, let us also illustrate a single drawback the virtualization yields—the performance overhead. Thus, to evaluate it together with the performance overhead introduced by the key forwarding mechanism required by the DiProNN, the quantitative evaluation of the Xen virtualization system follows in the latter section as well.

12.1 Qualitative Evaluation

This section presents the qualitative evaluation of the proposed programmable node architecture. We evaluate the objectives, that we have identified in the beginning of this thesis and discuss all the contributions the architecture provides.

VM-aware Execution Environment Architecture
This objective aims to improve the EEs' flexibility and security of such a programmable node. As already depicted, the employed virtualization makes the DiProNN able to run the processing applications (active programs) encapsulated in whole virtual machines, either provided and uploaded by users or provided by the node itself (the built-in VMs). The users are thus enabled to develop their applications for arbitrary[1] execution environments, while remain able to upload just standalone applications in cases, when these can make use of the provided built-in EEs.

The security improvements accrue from the node's ability to encapsulate the applications inside the VMs. First, the users are allowed to upload their applications encapsulated inside the VMs, which provide secure and reliable EEs they trust to—on the assumption, that the VMM behaves correctly, these EEs can be neither accessed by another users' applications nor by the node itself in other way than throught the network, which makes the running applications and/or data inside the EE better secured and less vulnerable. And even further, the node provides mechanisms to monitor and forbid the

[1] As obvious from the previous chapter, the EEs' flexibility depends on the virtualization system used by the particular DiProNN implementation.

undesirable communication among the running VMs, which makes the EEs vulnerable just throught the defined external interfaces (the DiProNN inputs), not by the co-running DiProNN applications.

Second, in the case of standalone applications making use of the built-in EEs, these are always provided with a fresh copy of the particular EE. On the one hand, this makes them able to obtain administrative privileges in the particular EE, while on the other they are ensured, that the particular EE has not been altered and/or compromised by a (malicious) application running there before. Moreover, if the particular implementation allows more applications to run inside a single VM, every EE is simultaneously shared just by the applications belonging to the same session[2]. Thus, the applications are still able to require administrative privileges in the particular EE, while they remain ensured, that another user's application, that has obtained the administrative privileges as well, cannot easily affect/compromise them.

The third security improvement must be considered from the node's point of view. Since the node's control plane, which runs inside a privileged VM (the Service Domain), is isolated from the users' applications as well, it cannot be affected/compromised by malicious users' applications, even thought these have received administrative privileges inside their EEs.

Even thought the common active/programmable nodes attempt to provide similar features, these are usually achieved as a trade-off between the programming flexibility and the architecture complexity. Against it, the VM-based systems have, in general, a significant security advantage over such traditional systems—the VMMs can be made simple[3], which can make them easier to verify and debug, and subsequently, less error-prone [134, 181] than conventional general-purpose operating systems, which the common active/programmable nodes usually rely on.

Nevertheless, as discussed in the motivation chapter, the employed virtualization also introduces a performance overhead, which is apparent especially on the I/O communication (see the tests presented in the following section). Thus, to cope with these issues we have sketched a design of a HW-accelerated network card suitable for its application in the DiProNN, which can reduce the network I/O overhead as low as possible. Unfortunately, the card has neither been designed in detail nor implemented yet, since the Liberouter project, with whom we have cooperated on the card's design, currently copes with higher-priority challenges (especially with finishing the NetCOPE platform, which the card relies on).

Component-based programming

The proposed programming model, which has been presented in the Chapter 6, assumes the applications consisting of several APs and data flows among them defined. As results from the previous paragraphs, the APs might be developed for any supported platform/EE and uploaded into the node either encapsulated inside a VM or not.

[2]Because of performance reasons, the particular implementation might decide to run applications from different sessions in the same EE. However, in this case, the particular EE has to monitor the proper applications' behavior, so that they cannot affect/compromise each other.

[3]The VMMs are relatively simple programs (for example, the Disco VMM has only 13 thousand lines of code [43], while, e.g., the Windows Server 2003 OS has about 50 million [182], the Fedora Core 3 OS about 70 million [13], and the Debian Etch OS even about 283 million [13] lines of code) with narrow, stable, and well-defined interfaces to the software running above it. Unlike traditional OSs, which have to support filesystems, network stacks, etc., the VMM "only" needs to present relatively simple abstractions, such as a virtual CPU and memory [97]. However, since many modern VMMs are much larger and more complex than it is required, they suffer from security vulnerabilities as well—see studies published in [81, 206].

The data flows among the APs are ensured using common network services—using standard network communication, which does not require the APs to support proprietary communication solutions/interfaces, and which further allows them to be distributed across several virtual machines and/or physical nodes/units. As soon as the APs' interfaces, which they want to communicate with, are associated with relevant network ports (see the static and dynamic ways of interfaces' association described in the Section 7.3.2), the data flows among them could be also ensured in two discussed ways—again, the static one and the dynamic one.

The DiProNN employs the dynamic flows' forwarding mechanism (all the APs send the data to a specific inter-mediator, which forwards the data to subsequent APs/receivers according to the defined communication channels), which allows the users to dynamically change the processing applications without any needs to restart them as well as which enables the inter-node VMs' migrations to perform high-level resource management. Even thought the interventions to the passing data streams are necessary, these do not introduce any overhead, as illustrated in the Section 12.2.4.

Possibilities of Parallel/Distributed Processing

As depicted within the motivation chapter, this objective aims to propose such a programmable node architecture, which allows to distribute both the processing and network load, so that it becomes able to process higher amounts of data in real-time. In DiProNN, such a load distribution is supported by both the parallel and distributed processing.

In terms of DiProNN, the parallel processing means a simultaneous processing of an intended active program onto several Processing Units. Once such a processing is required, the virtual machine, inside which the intended AP runs, is multiplied and deployed across specified number[4] of Processing Units. The distribution active program, which must precede every parallelizable AP, then distributes the incoming data over these parallel instances, which process them. From the user's point of view, there is no need to adapt the APs to support such a parallel processing; the only need is the nature of the data being processed, which have to be separable into independent data blocks, so that they can be spread over the parallel instances and processed independently on each other.

The parallel processing (and in general, any control communication among the APs) is further supported by possibilities of low-latency communication among the instances, which makes them able to synchronize and/or signalize any events. Again, the APs need not be adapted to utilize the low-latency interconnection in any way—once they register a particular communication channel as a control one, they are able to send the low-latency messages in the same way, as they do for the data ones.

The distributed processing in terms of DiProNN then means splitting of the intended application (the DiProNN Session), which consists of several APs, into multiple subsessions, which are independently processed on the relevant nodes in the network. Such a distribution can (along with the component-based applications' design) either allow the resource-demanding sessions to reach an additional computing power, which a single DiProNN would have been unable to provide, or ensure a more effective usage of the underlying network (see the "Fine-grained Resource Management System" subsection). However, the sessions cannot be splitted arbitrarily—as specified in the Section 8.1.3, the

[4]The number of parallel instances could be either fixed (specified by the user) or varying in time—the DiProNN is able to adjust the number of parallel instances depending on the actual amount of data, that are requested to be processed (see the Section 7.4.1.

scheduler, which decides about the distributed processing, must consider several constraints that have to be satisfied.

Fine-grained Resource Management System
The DiProNN's resource management system consists of a set of Resource Management modules, which run on the particular units, and which provide relevant information about the intended resources to the DiProNN Resource Management module, that manages the information about the resources throughout the node. As depicted in the previous chapter, the managed resources depend on the particular virtualization system, which is used for DiProNN's implementation.

The proposed node does not provide the resources to individual APs, as the common active/programmable nodes do. Rather, the resources are provided to individual VMs only—as described in the Section 3.1, this allows all the AP's and EE's processes, that belong to a particular user and/or imply from actions performed by his/her APs, to encapsulate into a single VM. From the VMM's point of view, such a VM then behaves as a single entity, which makes use of defined interfaces, through which it uses the provided resources. This makes the resource management and monitoring significantly easier and more precise [109] in comparison with traditional computing systems—as already depicted, the VMM provides significantly less interfaces and is considerably smaller and simpler than general-purpose OSs [58].

Nevertheless, if a fine-grained RMS—fine-grained in the sense, that the resources could be reserved for individual APs—is desired, one is still able to encapsulate each AP inside its own VM, which is further provided by the resources requested by that AP.

The encapsulation further allows the node to perform VMs' migrations—moving a VM from one physical host to another. Such migrations are an essential feature for a high-level resource scheduling—a VM could be migrated to a different DiProNN's unit because, e.g., a less used unit is available, or because there is a need to gather the less-demanding VMs, so that a resource-consuming one(s) could be deployed. Besides such intra-node migrations, the DiProNN supports the inter-node migrations as well—moving a VM from one node to another. These migrations, which are enabled by the employed dynamic data forwarding mechanism, could be beneficial either from similar reasons, as the intra-nodes migrations are, or because of better and/or more effective usage of the network (e.g., the session distributes the incoming data and most receivers are located nearer a different node). In both cases, the principle called *live migrations* [64] could be used—a VM is migrated without any needs to pause it, so that the end-user applications need not notice them (except situations when the user wants to be informed about such events).

Besides the resource reservations, the employed virtualization allows the DiProNN to support VM-based resource guarantees as well. However, these require the particular virtualization system to provide powerful scheduling algorithms satisfying several assumptions, which have been discussed in the Section 9.1.1. Moreover, in the case of distributed sessions' processing, the DiProNN's resource guarantees have to be supported by appropriate services, which are able to guarantee common networking parameters on the public networking infrastructure.

Flexible Data Transmission Protocol Architecture
As already discussed in the Section 3.2.5, the common active/programmable nodes usually operate just on top of the Network layer of the ISO/OSI Network model and include all the necessary service information, which (among others) serves for distinguishing

among the sessions and their flows, just behind the packets' IP header. This makes them able to support arbitrary pure transport protocols.

As opposed to it, the DiProNN does not insert any service information into the packets. On the one hand, this does not introduce any overhead related to wasting the bandwidth on the service information, but on the other, this requires the DiProNN to rely on an information provided by the transport protocols used—in particular, on their multiplexing feature discussed in the Section 5.1. Otherwise, the node would have been able neither to distinguish among the sessions' data flows nor to ensure proper communication channels among the APs.

This requires the transport protocols, that the DiProNN users want to use for their applications, to be supported by the node. Nevertheless, this restriction relates just to the pure ones, since the application-level ones, which make use of an underlying supported pure transport protocol, are not obviously limited.

12.2 Quantitative Evaluation

As the Chapter 10 depicts, there are many applications, that might be performed on the active/programmable nodes like the DiProNN is. In general, these applications could be divided into the CPU intensive applications (the ones, that require/produce a negligible amount of input/output data, above which they perform huge computations), the CPU and I/O intensive applications (the ones, that besides the computations require/produce a reasonable amount of input/output network data or that perform a reasonable number of disk operations), and the I/O intensive applications (the ones, that require/produce huge amount of input/output network data or that perform many disk operations).

To illustrate the real virtualization's performance overheads for each mentioned kind, we have performed several tests that are presented in the following sections. Moreover, the overhead introduced by the forwarding mechanism, which the DiProNN relies on, is analyzed in the Section 12.2.4 as well.

12.2.1 Experimental Setup

In order to perform the evaluation, we have set up a testbed consisting of two physical machines directly interconnected with 10 Gigabit Ethernet link. The hardware and software configuration of the machine, that has run the Xen virtualization system, is described in the Table 12.1 and 12.2 respectively. The machine has been set in the dual-boot mode—one system has run the Xen[5] virtualization system (version 3.4.0), whilst the other has run native Linux OS (for the purposes of better expressiveness, with the same kernel version as the Xen system has run).

The hardware and software configuration of the other machine, that has been used for network measurements as a data generator and analyzer, is described in the Table 12.3.

12.2.2 CPU and disk I/O Performance Overhead Tests

12.2.2.1 Test's methodology

As already depicted, the goal of this test is to check, whether the virtualization introduces an overhead for the CPU and/or disk I/O intensive applications. For the tests, we have

[5] `http://www.xen.org/`

	Configuration
Motherboard	SuperMicro X7DBR-i Rev: 1.01
Processor	2× Dual-Core Intel Xeon 3.0 GHz
Memory	4 GB (Kingston 2RX4 PC2-5300F)
10GE NIC	Myricom Myri-10G, type 10G-PCIE-8A firmware 1.5.0, jumbo frames enabled

Table 12.1: Hardware configuration of the VM host node.

	Native Linux	*Xen (version 3.4.0)*	
		dom0	*domUs*
Operating system	Ubuntu 7.04 (Feisty Fawn)	Ubuntu 7.04 (Feisty Fawn)	Ubuntu 7.04 (Feisty Fawn)
Kernel version	2.6.18 SMP	2.6.18 SMP	2.6.18 SMP
CPUs	4	2	1
Memory	4 GB	2 GB	512 MB

Table 12.2: Software configuration of the VM host node.

used two applications with different proportion of the CPU and disk I/O-intensive behavior (namely, the *POV-Ray*[6] and the *Gaussian*[7] [92]) and measured the time necessary for computing the results under various conditions.

In order to evaluate the overhead, we have run the applications in the native Linux OS at first. After that, the same tests have been run in a Xen's VM, where—besides the time necessary for computing the results—we have also measured[8] the CPU load of the Service Domain (so-called *dom0*) to observe, how much does the VM's CPU load affect the *dom0*.

[6] `http://www.povray.org/`

[7] `http://www.gaussian.com/`

[8] For the CPU's load measurements, the *xentop* tool, which is a standard Xen's replacement of the *top* command, has been used.

	Configuration
Motherboard	TYAN Thunder n6650W (S2915)
Processor	2× Dual-Core AMD Opteron 2.6 GHz
Memory	4 GB (Kingston KVR667D2S4P5/1G)
10GE NIC	Myricom Myri-10G, type 10G-PCIE-8A firmware 1.5.0, jumbo frames enabled
Operating system	Ubuntu 7.10 (Gutsy Gibbon) kernel 2.6.24 SMP

Table 12.3: Hardware and software configuration of the generator/analyzer machine.

During the tests, the Xen's primary *Credit scheduler*[9] has been used—the *weight* and *cap* values for both the *dom0* and the VM have been set to default values (256, resp. 0). All the tests have been performed three times—the results, that are summarized in the Table 12.4, then represent the average values measured.

POV-Ray
The *POV-Ray (Persistence of Vision Raytracer)* is a software tool originally dedicated for creating three-dimensional graphics. Nevertheless, it is often used for CPU benchmarking as well, since it performs negligible number of I/O operations, but performs huge computations—there is a standard scene (`benchmark.pov`) explicitly intended for such a benchmarking.

For the tests, we have used the version 3.5 of the intended benchmark and using the `benchmark.ini` file we have instructed the POV-Ray to compute the scene of the size 640x480 pixels. The version of the POV-Ray, that has been used, was 3.6.1.

Gaussian
The *Gaussian* is a package of programs based on basics of quantum mechanics. It originally serves to predict the energies, molecular structures, and vibrational frequencies of molecular systems, along with numerous molecular properties derived from these basic computation types. Similarly to the POV-Ray, it is also often used for the benchmarking.

The presented tests have been performed using the Gaussian version G03.E01. We have used two supplied test files—the `test497`, which computes NMR chemical shifts of a small molecule, and the `test540`, which computes forces affecting the nuclei as well as vibration frequencies. Whilst the `test540` is mainly CPU-intensive performing a low number of I/O operations (it produces about 68 MB of temporary data during the test), the `test497` creates lots of temporary files and thus behaves both CPU & I/O-intensive (about 1.4 GB of temporary data is saved and further read during the test).

		native Linux	*Xen*	
		runtime [sec]	*runtime [sec]*	*dom0 CPU load [%]*
POV-Ray		1575	1576	0.84
Gaussian	`test497`	356	423	9.94
	`test540`	1087	1108	1.67

Table 12.4: The comparison of tested applications' runtimes within virtualized and non-virtualized environments.

12.2.2.2 Results discussion

The results, which are summarized in the Table 12.4, indicate, that in the case of pure CPU-intensive applications, the virtualization (Xen) introduces a negligible overhead as compared to the native OS. Nevertheless, once the number of I/O operations increases, the overhead is more perceptible—it is evident on both the Gaussian's increased runtimes

[9] `http://wiki.xensource.com/xenwiki/CreditScheduler`

as well as on the increased Service Domain's CPU load[10], which has to intervene every I/O operation. In the case of I/O intensive applications, this overhead significantly influences the applications' performance (for more details, see the tests published in [60]).

12.2.3 Network Performance Overhead Tests

12.2.3.1 Test's methodology

This test aims to illustrate the overhead introduced by processing the network flows in the Xen VMM. Even thought there have been many Xen's network performance-related studies already published (e.g., [18, 19, 60, 191, 281]), as far as we know, no one has performed the measurements using the 10 Gigabit Ethernet network with jumbo frames[11] enabled yet. Thus, since such an interconnection is the advisable one for the DiProNN's internal data interconnection, we have performed a set of tests illustrating the achievable network throughputs in such an environment.

The overhead has been measured both for the flows passing throught the user domain[12] as well as for the one-way ones (again, both in the native Linux OS and the Xen VMM). The measurements cover the pure network throughputs without any additional computations performed. For the measurements covering the network performance combined with intensive CPU computations under various conditions (different schedulers, different proportions of CPU times assigned to the domains), let us refer you to our comprehensive measurements published in [226] or to the measurements published in [61].

To measure the throughput, latency, and jitter, the tool called *generator7* has been used. The *generator7* is a UDP packet generator and receiver based on our previous implementation of the *RTPgen/RTPsink* toolset [120]—the *generator7* uses RTP-like time-stamps and sequence numbers to measure the bandwidth, packet loss, latency, and jitter of the communication (the jitter is measured according to the methods described in [236]). Similarly to the previous measurements, the *dom0*'s CPU load has been monitored in one-second intervals using the *xentop* tool. For each measured packet size[13] we have been exploring the maximal throughput, that could be achieved without any losses in the data stream. All the tests have lasted 60 seconds and have been performed three times—again, the presented results represent the average values measured.

During the measurements, the Xen has been set in the *routed* network mode, the Credit scheduler with default *weight* and *cap* values for both the running domains (*dom0* and measured VM) has been used, and the network buffers have been set to 16 MB. As suggested[14] by Myricom, to achieve better performance the 10 GE NIC's interrupt coalescing value has been further increased[15] to 200 μs (the default is 75 μs).

[10] Note, that the *dom0*'s CPU load measurements have been influenced by the load monitoring itself, which has been performed within the Service Domain.

[11] For the measurements, we have set the MTU to 9000 B and measured the throughputs for the packets up to 8800 B of size (not including the IP and Ethernet headers).

[12] In the case of passing throughput measurements, the data flows have been returned to the sending machine by a simple application-level tool, that we have created.

[13] In the range from 100 B to 1500 B, the measurements have been performed in 100 B intervals, while in the range from 1500 B to 8800 B in 200 B intervals.

[14] `http://www.myri.com/serve/cache/570.html`

[15] Command: `ethtool -C <device> rx-usec <value>`

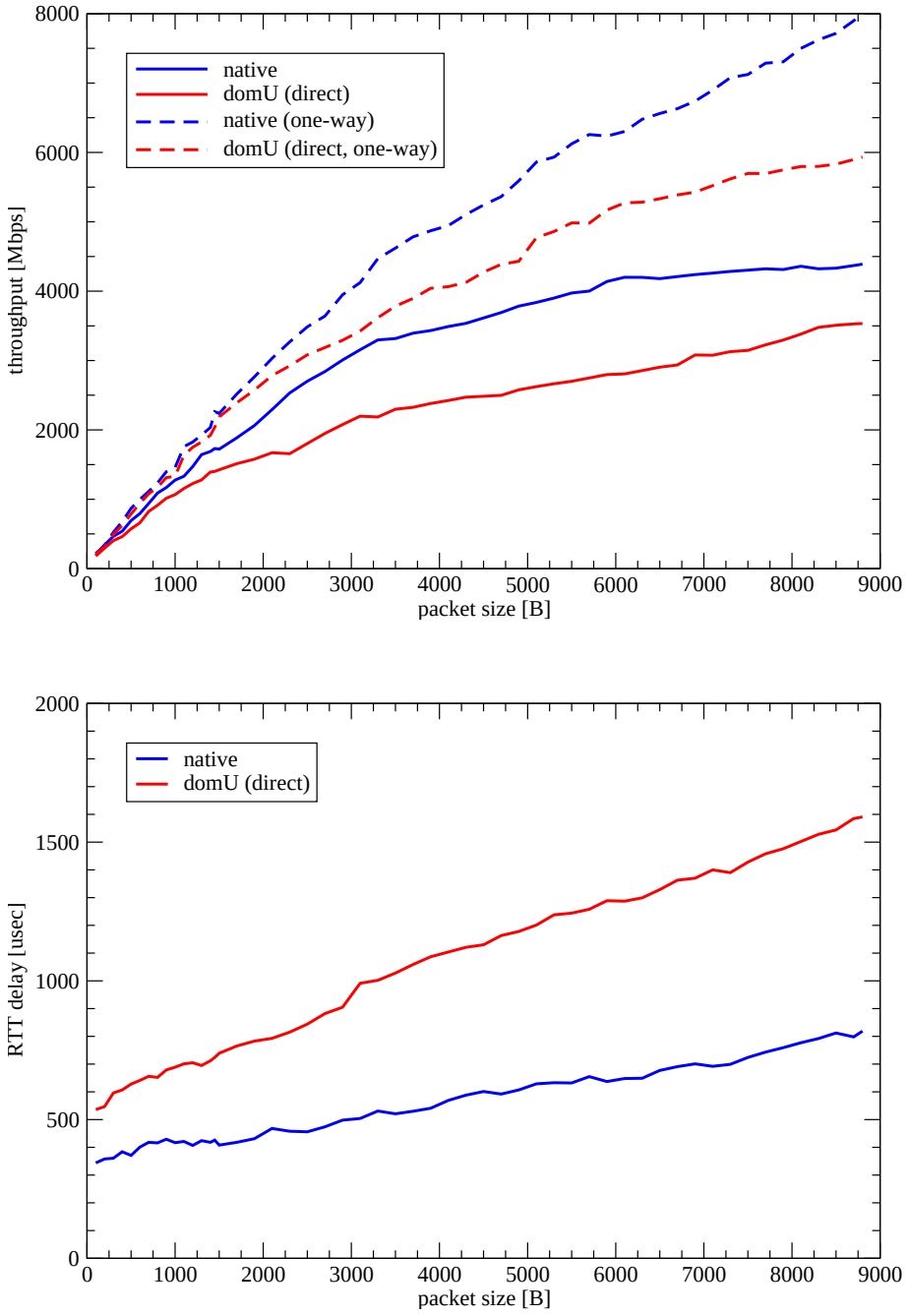


Figure 12.1: The comparison of network performance in the native and virtualized systems (throughput and delay).

12.2.3.2 Results discussion

The graphs in the Figures 12.1 and 12.2 show the maximal throughputs and corresponding RTT delays and jitters achieved during the measurement. In the case of one-way flows, the graphs show just the average values of the results, that have been measured for both directions, and because of desynchronized clocks on both the machines, the RTT delay and jitter values are presented just for the bi-directional flows.

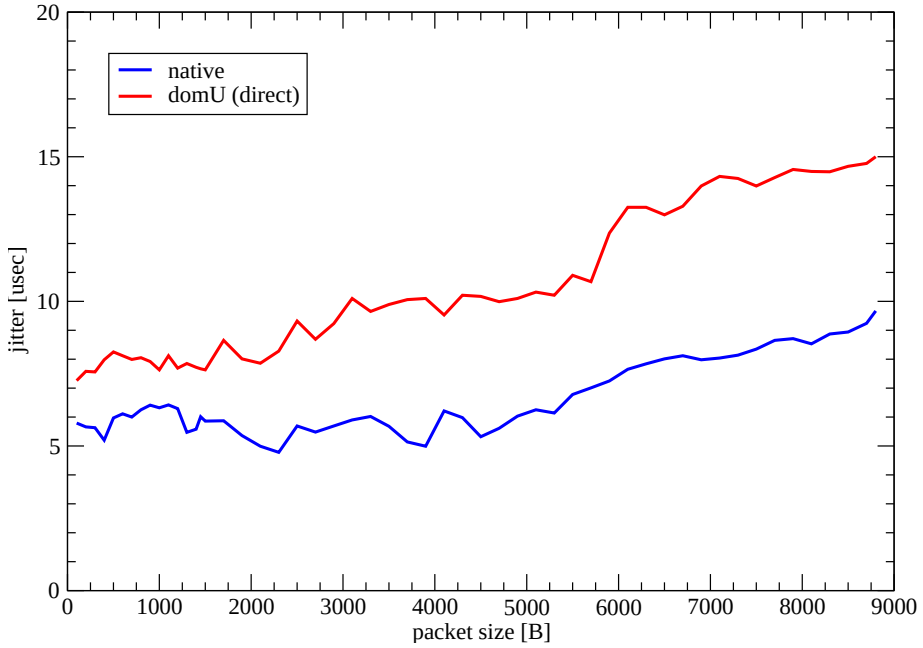


Figure 12.2: The comparison of network performance in the native and virtualized systems (jitter).

The results indicate the significant loss in throughput and corresponding increase of the end-to-end latency in the case of the virtualized system's test. Moreover, the virtualization's overhead can be further observed in the high utilization of the *dom0*, which has to handle all the packets within the VMM (see the Figure 12.4).

As we have already depicted, even thought the overhead is rather high, we have decided not to restrict the DiProNN's design to current issues in the virtualization systems, since we believe, that they will be solved in close future. Moreover, if one requires higher network performance, another virtualization system could be used for its implementation—for example, an OS level one which on the other hand, however, leads to lowering the node's EEs flexibility.

12.2.4 DiProNN Forwarding Mechanism's Tests

12.2.4.1 Test's methodology

In this case, the performance overhead of the DiProNN's forwarding mechanism performed by the kernel *Netfilter*[16] (also known as `iptables` [15]) has been studied. The tests have been performed in exactly the same scenario as the ones being described in the previous section, however, the flows outgoing the VM(s) have been directed to the *dom0*, where they have been forwarded to the receiver according to the defined `iptables` rules[17].

[16] `http://www.netfilter.org/`

[17] Nevertheless, in the case of one-way tests performed in the direction from the sender/analyzer machine to the VM, the flows have been obviously forwarded in the opposite direction.

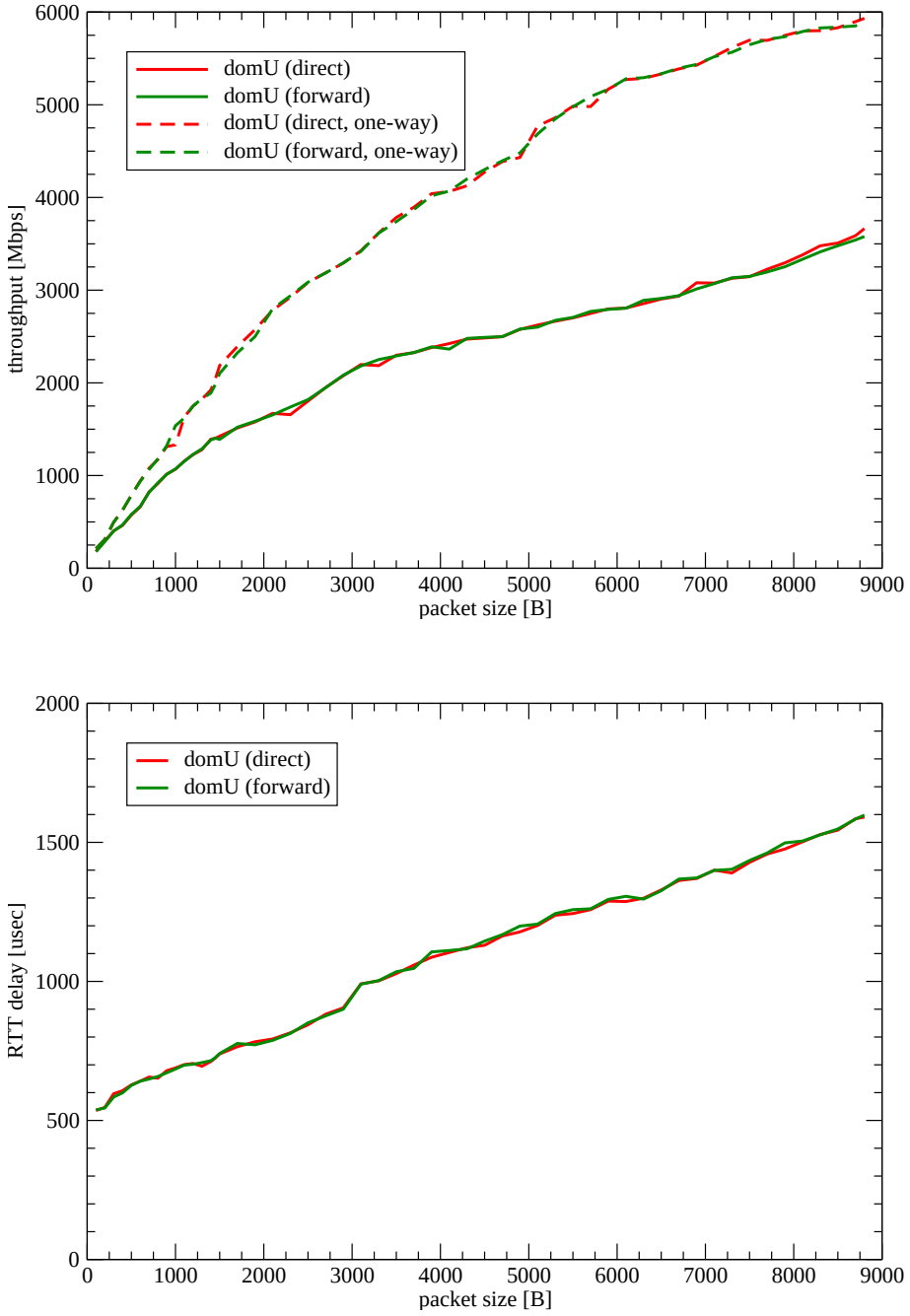


Figure 12.3: The comparison of network performance in the case of direct access and `iptables` forwarding (throughput and delay).

12.2.4.2 Results discussion

The Figures 12.3 and 12.4 clearly indicate, that there is no appreciable performance overhead introduced by the use of the Netfilter—neither the bandwidth measurements, nor the timing ones indicate any degradation of the flows' processing (in fact, except of a bit smoother behavior). The achieved results conform to more comprehensive Netfilter studies [124, 219] performed on native Linux OSs.

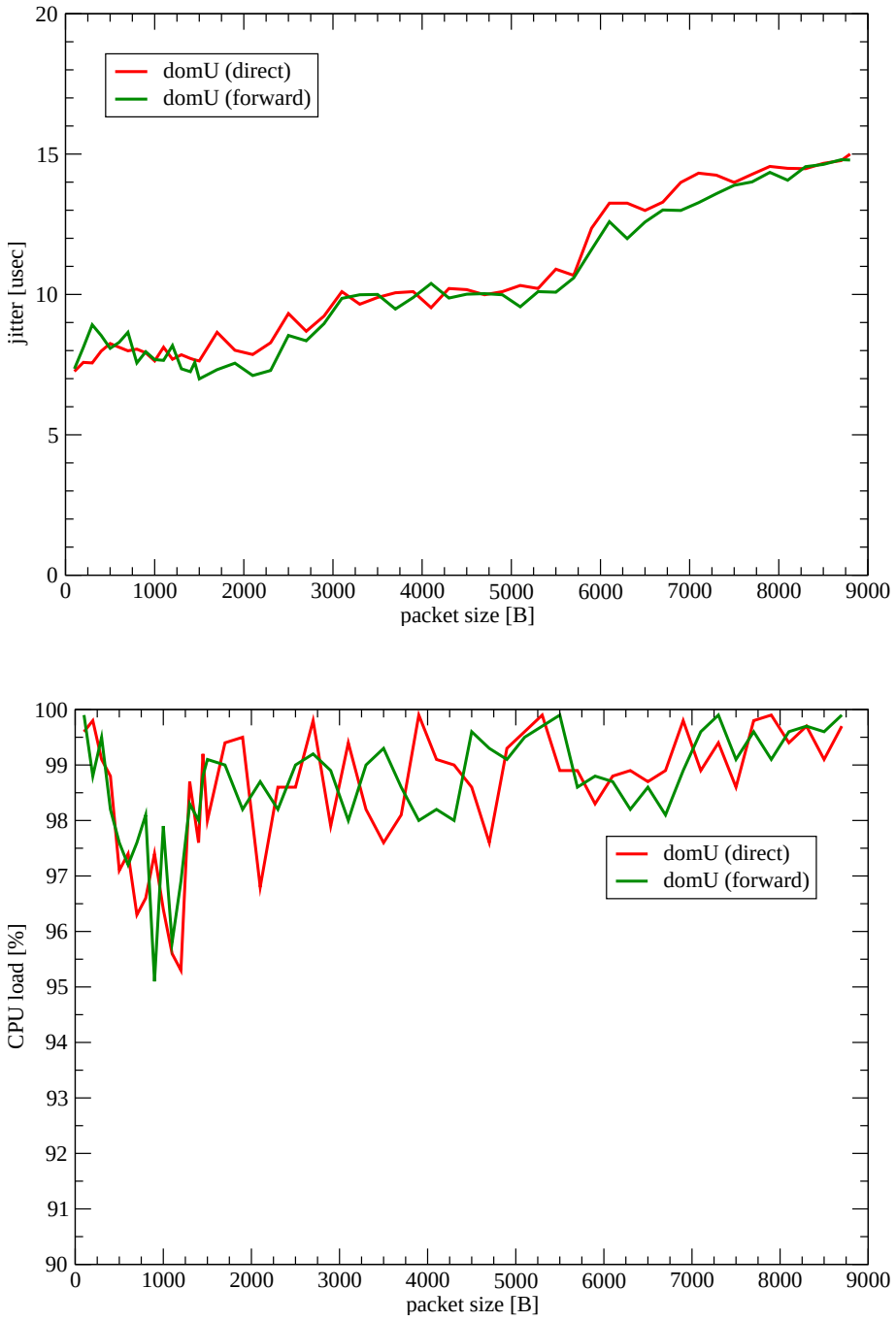


Figure 12.4: The comparison of network performance in the case of direct access and `iptables` forwarding (jitter and *dom0*'s CPU load).

The CPU load of the *dom0*, which performs the forwarding itself, has been monitored as well. Nevertheless, neither this measurement indicates any affects on the *dom0*'s load (anyway, if there are some, these are hidden in the fluctuations related to the computations, that are required by handling the VM's network flows).

Chapter 13

Usage Example

To illustrate the possibilities of DiProNN's usage, we have built a processing infrastructure consisting of four nodes located in Brno, Liberec, Pilsen, and Prague (see the Figure 13.1). The nodes, which have been set using the DiProNN's minimal architecture possible, and whose configuration is depicted in the Table 13.1, have been interconnected with 1 GE network links provided by the public networking infrastructure.

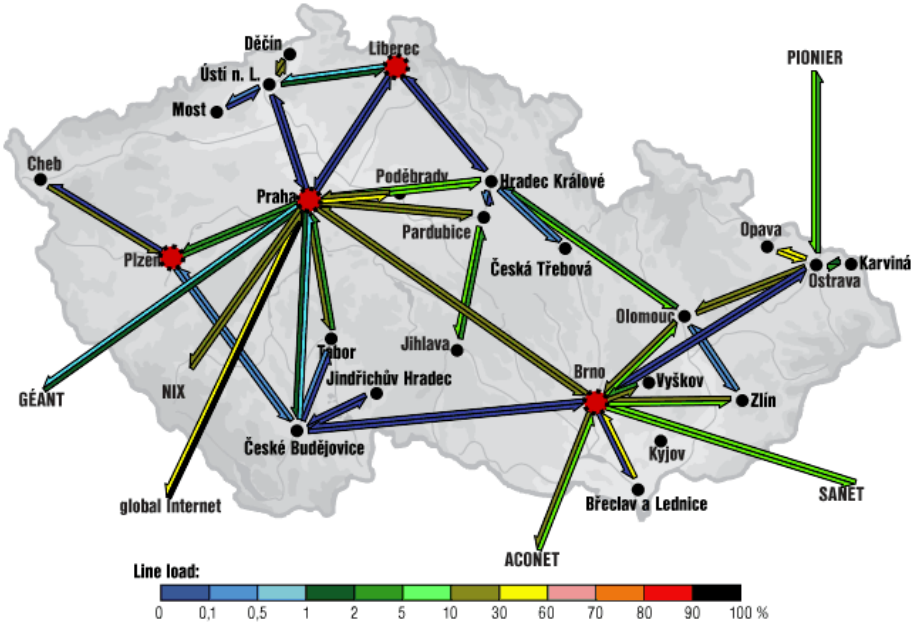


Figure 13.1: The DiProNN nodes used for the processing infrastructure (red dots).

The infrastructure is intended to provide a multimedia stream distribution and processing platform for applications required by the "Multimedia transmissions and collaborative environment" research group of the Cesnet association. However, since the service utilities, which serve for nodes' configuration and sessions' establishments (uploading, starting, setting, and destroying the APs/VMs), have not been implemented yet, the whole configuration has been performed manually.

	Configuration
Brand	Supermicro
Model	X7DBR-i Rev: 1.01
Processor	2× Dual-Core Intel Xeon 3.0 GHz
Memory	4 GB DIMM DDR2
GE NIC	2× Intel PRO/1000 Network Adapter
Operating system	Linux Ubuntu 7.04 (Feisty Fawn)
	Xen version 3.4.0
	kernel 2.6.18-xen SMP

Table 13.1: Configuration of the DiProNN nodes, which the processing infrastructure is built from.

13.1 Sample Scenario

Situation: To demonstrate a potential usage, we have simulated a presentation taking place in Brno, which had to be also available for the auditors not being able to attend it personally. For the clients with a high-bandwidth network connection, the presentation had to be available in a high-quality HDV stream (generated in Brno using an HDV camera[1]), while for the clients located behind low-bandwidth network lines, it had to be transcoded into a lower quality in the real-time. Both the high-quality and low-quality streams had to be saved for later purposes as well.

Since the video transcoding takes some time, at least a synchronization of the audio[2] and transcoded video streams must be performed. Nevertheless, to prevent network fluctuations and thus their possible desynchronization, the synchronization of the original audio and video streams is also very desirable.

However, it is important to point out, that the synchronization of the original audio and video streams has to be performed because of one another reason: the HDV stream outgoing the HD camera has been delayed by approximately 1 second by the camera itself, while the audio stream has been captured by a standalone audio grabber device with the latency in order of tens of ms. Thus, both these streams have had to be synchronized, even if the network itself had not desynchronized them.

For the described situation, we have established a session, whose DiProNN Session Graph and relevant DiProNN Program are depicted in the Figures 13.2 and 13.3. Both the audio and video streams have been sent to the node located in Brno (the `V_in` input for the video stream and the `A_in` input for the audio stream), where they have been duplicated (the `Dup_A` AP for the audio stream's duplication and the `Dup_V` AP for the video stream's duplication). One *twinstream* (audio and video) has been sent to the Prague node (the high-quality one), while the second one has been forwarded to the second VM running on the Brno node for the transcoding (the `Transcode` AP), and afterwards sent to the Liberec node. Subsequently, both twinstreams have been synchronized (the `Sync_high` and `Sync_low` APs) and duplicated once again (the `Dup_high` and `Dup_low` APs). These latter twinstreams have been sent to the Pilsen node and saved

[1] For our experiments, we have used the Sony HVR-Z1R camera.

[2] As results from the composed DiProNN Session, the audio stream was not transcoded during the experiment, since it does not require as high network bandwidth as the video stream does.

for later purposes (the `Saver_high` and `Saver_low` APs), and to the reflector applications [121], which have served as content providers for the clients (both of them have provided a single DiProNN input port for registering the clients and two DiProNN output ports for sending the data).

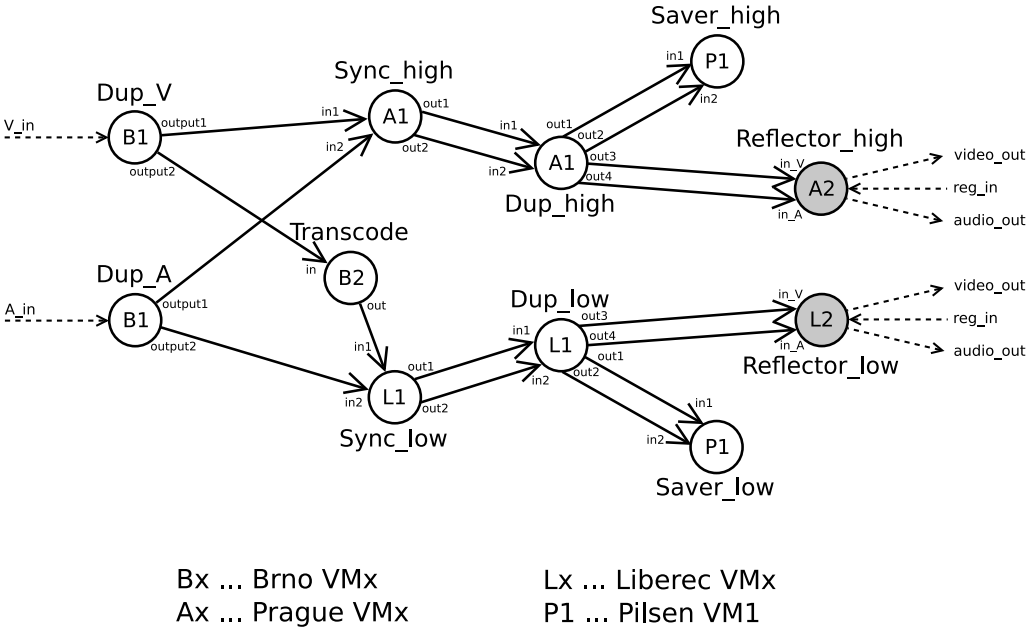


Figure 13.2: The DiProNN Session Graph describing the example scenario.

The original video stream has been transferred in the HDV format [101], which has taken about 25 Mbps of the network bandwidth. It has been captured, played as well as transcoded by the VLC media player[3]—regarding the transcoding itself, the input HDV stream has been transcoded into the MP2V format with a variable bitrate (set to 256 Kbps) and scaled down to 25 % of its original size. The audio has been captured and played by the RAT application[4], which has been set to the Linear-16 codec and the 16 KHz stereo mode (thus taking 512 Kbps of the network bandwidth). Both streams have been synchronized using our application presented in [78], and saved in a simple packet form by a saving application[5], which we have created during our previous experiments.

The overall client's setup (located in Brno) is captured in the Figure 13.4, while a detail of client's screen is captured in the Figure 13.5. The client has been connected to both the high-quality and low-quality content providers, and thus the figure further shows the high-quality stream's bandwidth (which has taken about 30 Mbps including the packets' headers), the low-quality stream's bandwidth (about 820 Kbps including the packets' headers), and the latency introduced by the transcoding itself (visible on the streamed clocks—roughly about 1 second). The sizes of the saved files containing 20 minutes of the streams, have been 3,2 GB for the high-quality video stream, 103 MB for the transcoded video stream, and 79 MB for the both the high-quality and "low-quality" audio streams (as depicted before, the audio stream has not been transcoded).

[3] `http://www.videolan.org/vlc`

[4] `http://mediatools.cs.ucl.ac.uk/nets/mmedia`

[5] The application saves the whole UDP packet's content together with a timestamp information into a file. We have also created a player capable of reading such a file content and sending it to the network like it would have been sent in the real-time.

```
# some session's parameters
{ AP name="Dup_V" ref="DiProNNservice.duplicator";
     inputs = V_in(DIPRONN_INPUT(10000));
              # requested DiProNN video input port is 10000
     outputs = output1(Sync_high.in1), output2(Transcode.in);
}
{ AP name="Dup_A" ref="DiProNNservice.duplicator";
     inputs = A_in(DIPRONN_INPUT(10002));
              # requested DiProNN audio input port is 10002
     outputs = output1(Sync_high.in2), output2(Sync_low.in2);
}
{ AP name="Transcode" ref="DiProNNservice.transcoder";
     inputs = in;
     outputs = out(Sync_low.in1);
     output_format = "mp4v";
     bitrate = "variable(256)";
     scale = "0.25"
}
{ AP name="Sync_high" ref="DiProNNservice.syncer";
     inputs = in1, in2;
     outputs = out1(Dup_high.in1),out2(Dup_high.in2);
     precision = 0.001;  # 1ms
}
{ AP name="Sync_low" ref="DiProNNservice.syncer";
     inputs = in1, in2;
     outputs = out1(Dup_low.in1),out2(Dup_low.in2);
     precision = 0.001;  # 1ms
}
# Dup_high and Dup_low defined similarly as Dup_V and Dup_A
...
{ AP name="Saver_high" ref="DiProNNservice.saver";
     inputs = in1, in2;
     output_file = "stream_high.dump";
}
{ AP name="Saver_low" ref="DiProNNservice.saver";
     inputs = in1, in2;
     output_file = "stream_low.dump";
}
{ VM ref="my_VM.img";
    { AP name="Reflector_high" ref="reflector";
         inputs = in_V, in_A, reg_in(DIPRONN_INPUT(12345));
         outputs = video_out(DIPRONN_OUTPUT(PASSTHRU)),
                   audio_out(DIPRONN_OUTPUT(PASSTHRU));
    }
}
{ VM ref="my_VM.img";
    { AP name="Reflector_low" ref="reflector";
         inputs = in_V, in_A, reg_in(DIPRONN_INPUT(12354));
         outputs = video_out(DIPRONN_OUTPUT(PASSTHRU)),
                   audio_out(DIPRONN_OUTPUT(PASSTHRU));
    }
}
```

Figure 13.3: The DiProNN Program describing the example scenario.

Figure 13.4: Client side setup.

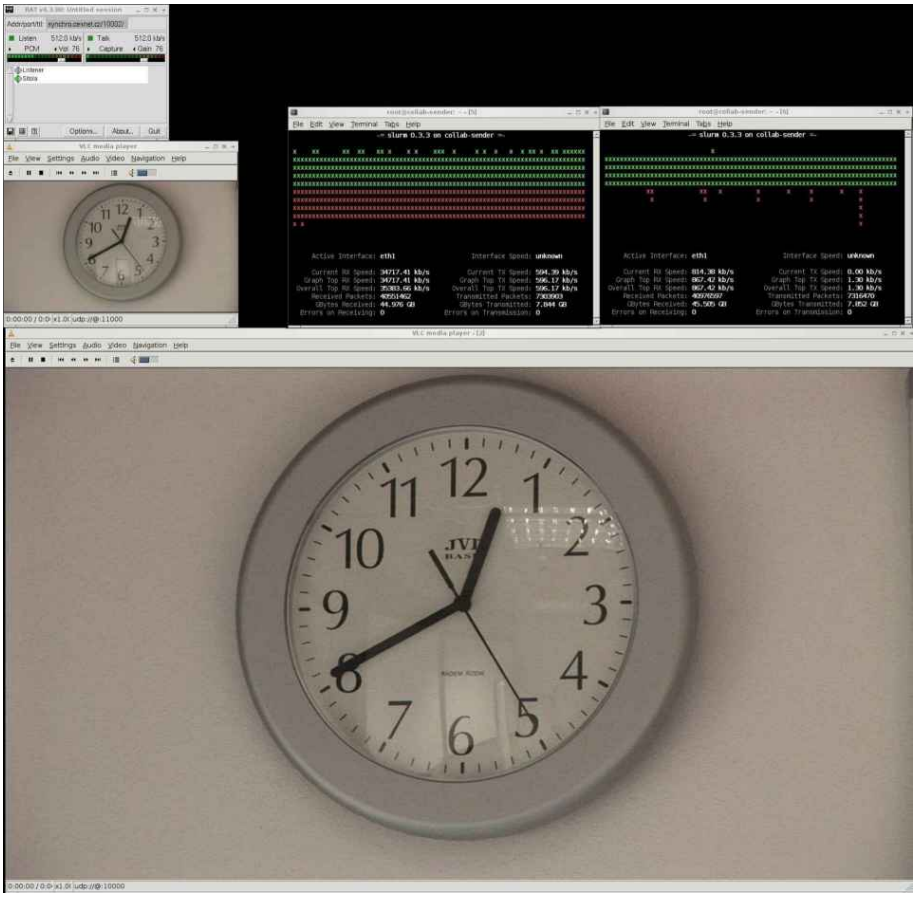


Figure 13.5: Client side's screenshot. The clocks' difference shows the latency introduced by the transcoding itself (roughly about 1 second).

Chapter 14

Conclusions

Even thought the primary idea behind the active/programmable networks was to enable a fast deployment and customization of novel network services, throughout the years, they have also emerged into a suitable mechanism for intra-network stream processing services—for the services, that require some (usually real-time) computations to be performed over passing data streams, which cannot be achieved and/or are not efficient on the end-hosts. Nevertheless, in the recent years, the active/programmable networks are slightly losing momentum—many research teams have moved their focus into another areas, more or less related to the processings in the network. This fact is not implied by a lose of interest for the intra-network stream processing—the needs for an available stream processing power, which is accessible within the network for the end-users, still remain [1, 99, 249, 304]—but, as we assume, especially by their insufficient flexibility in terms of execution environments, that the proposed nodes are able to provide (the users are forced to create their active applications only for defined, usually highly specialized EEs), and weak security guarantees (as we have mentioned previously, the security guarantees are, if any, usually achieved to the prejudice of the programming flexibility).

In this thesis, we have adduced the possibilities of improving these weaknesses. We claim, that the use of virtualization can not only enrich the flexibility of the active/programmable nodes' EEs, but can further provide another useful features, like strong isolation among the running applications and simpler attainability of robust security guarantees (both from the users' and nodes' point of view). Moreover, since the users are strictly isolated within the EEs running in the VMs, they could be provided with administrative privileges without any degradation of nodes' security.

The presented benefits have been illustrated on a novel VM-aware programmable network node, named DiProNN, which combines the use of virtualization with another useful concepts—namely, with the component-based programming and the parallel/distributed processing. The proposed node, whose architecture is presented in the Chapter 4, thus provides a powerful processing infrastructure, which allows its users to develop the active programs for arbitrary execution environments and comfortably compose them into complex processing applications. The data flows among the components (APs) are ensured using standard network communication, which (among others) allows the users to make use of legacy applications without any needs to perform essential changes into them. Such applications' separation into multiple individual components combined with the standard networking communication model further allows the DiProNN to provide a significant processing power, since it is able to distribute the processing load of a single application onto multiple nodes in the network and/or multiple processing elements within a single node, as described in the Chapter 8.

Besides the benefits, the performance overhead introduced by the virtualization has been studied as well. We have performed several tests, which aim to illustrate the real overheads of the characteristics, which we consider to be the most important for the active/programmable nodes—the computing overhead, the overhead of processing the applications with different proportion of the CPU and I/O-intensive behavior, and the overhead of processing the network flows. The achieved results confirm the expected behavior—whilst the overhead of processing pure CPU-intensive applications is negligible, as soon as the amount of I/O operations increase, the overhead increases as well.

Such an increased I/O overhead also pertains to the processing of network flows, which are obviously the main factor of the active processing. Even thought the introduced overhead is not serious to make the virtualization entirely unusable for the active/programmable network nodes and even thought more powerful virtualization systems could be used for the implementation (especially the OS-level ones, which however, slightly limit DiProNN's features), we have also sketched the architecture of an FPGA-based programmable hardware network card (the Section 9.2), which accelerates both the employed forwarding mechanism and the whole network stack, and which thus can be assumed to have a potency to lower the network processing overhead as low as possible. Nevertheless, there are also many other projects trying to analyze and improve the software parts of the virtualized network stacks (especially, within the Xen VMM [19, 191, 281]).

Concerning the future challenges, besides the ongoing efforts to complete the DiProNN's prototype implementation (especially in the sense of automated processing of sessions' establishments, so that the node becomes usable without any needs for manual configurations), our further work will focus on establishing the infrastructure depicted in the previous chapter in order to provide a fully automated, flexible, and powerful computing service in the network. Such a service is aimed to serve for various projects related to the Cesnet's "Multimedia transmissions and collaborative environment" activity—for example, the HD and/or 4K video streams' processing (e.g., real-time transcodings, stereoscopic synchronizations, and/or DXT compressions [269]) and distribution.

List of Abbreviations

ABI	Application Binary Interface
AFS	Andrew File System
AI	Artificial Intelligence
AMD	Advanced Micro Devices
ANAP	Active Node Authentication Protocol
ANEP	Active Network Encapsulation Protocol
ANN	Active Network Node
ANPE	Active Network Processing Element
ANTS	Active Network Transport System
AP	Active Program
API	Application Programming Interface
APM service	Active Program Manager service
AR	Active Router
ARM	Advanced RISC Machine
ARTP	Active Router Transport Protocol
ATM	Asynchronous Transfer Mode
BIOS	Basic Input-Output System
BVT	Borrowed Virtual Time scheduler
C	C Programming Language
CAML	Categorical Abstract Machine Language
CIL	Common Intermediate Language
CISC	Complex Instruction Set Computer
CLARA	CLuster-based Active Router Architecture
CLI	Common Language Infrastructure

CLR Common Language Runtime

CORBA Common Object Requesting Broker Architecture

CPU Central Processor Unit

DAG Directed Acyclic Graph

DAN Distributed Code Caching for Active Networks

DARPA Defense Advanced Research Projects Agency

DCCP Datagram Congestion Control Protocol

DCCP Datagram Congestion Control Protocol

DDR Double Data Rate

DFS Distributed File System

DHCP Dynamic Host Configuration Protocol

DiProNN Distributed Programmable Network Node

DPE Distributed Processing Environment

DSP DiProNN Scheduling Problem

EE Execution Environment

EGRE Ethernet over GRE

FAIN Future Active IP Networks

FPGA Field-Programmable Gate Array

FPGA Field-Programmable Gate Array

FRTP Fixed Rate Transport Protocol

FTP File Transfer Protocol

FTPS FTP Secure Protocol

Gbps Gigabit per second

GE Gigabit Ethernet

GLUE Grid Laboratory Uniform Environment

GPL General Public License

GPU Graphics Processing Unit

HD High-Definition

HDV High-Definition Video

HPC High Performance Computing

HW	Hardware
I/O	Input/Output
IDS	Intrusion Detection System
IP	Internet Protocol
IR	Intermediate Representation
ISA	Instruction Set Architecture
IST	Information Society Technologies
JANOS	Java-oriented Active Network Operating System
JDK	Java Development Kit
JSDL	Job Submission Description Language
KVM	Kernel-based Virtual Machines
LARA	Lancaster Active Router Architecture
LARA++	Lancaster's 2^{nd}-generation Active Router Architecture
LARA/MAN	LARA MANagement component
LARA/PAL	LARA Platform Abstraction Layer
LARA/RT	LARA Run-Time execution environment
LLVM	Low Level Virtual Machine
MAC address	Media Access Control address
Mbps	Megabit per second
MCU	Multipoint Control Unit
MIB	Management Information Base
MIPS	Microprocessor without Interlocked Pipeline Stages
MIT	Massachusetts Institute of Technology
MKP	Multiple Knapsack Problem
MP2V	Media Excel MPEG-2 Video codec
MPI	Message Passing Interface
MPLS	MultiProtocol Label Switching
MS	Microsoft
MSSP	Multiple Subset Sum Problem
MTU	Maximum Transmission Unit

MU	Multimedia Unit
NACK	Negative ACKnowledgment
NAT	Network Address Translation
NETBLT	NETwork BLock Transfer Protocol
NFS	Network File System
NIC	Network Interface Card
NodeOS	Node Operating System
NS	Network Simulator
NVN	NetScript Virtual Network
OS	Operating System
PA service	Port Associator service
PDF	Portable Document Format
PDU	Protocol Data Unit
PKI	Public Key Infrastructure
PLAN	Packet Language for Active Networks
POV-Ray	Persistence of Vision Raytracer
procfs	Process File System
QDR	Quad Data Rate
QM	Query Machine
QoS	Quality of Service
QsNet	Quadrics Network
QuaSAR	Quality of Service Aware Router
RAM	Random-Access Memory
RAT	Robust Audio Tool
RFC	Request For Comments
RISC	Reduced Instruction Set Computer
RMS	Resource Management System
RMTP	Reliable Multicast Transport Protocol
RPC	Remote Procedure Call
RSVP	Resource ReSerVation Protocol

RTP Real-time Transport Protocol

SAGE Scalable Adaptive Graphics Environment

SANE Secure Active Network Environment

SCP Secure Copy Protocol

SFTP Secure File Transfer Protocol

SSA Static Single-Assignment

SSD Solid State Disk/Drive

SSH Secure SHell

SW Software

SwissQM Scalable WIrelesS Sensor Query Machine

TCP Transmission Control Protocol

TINA Telecommunications Information Networking Architecture

UDP User Datagram Protocol

UDT UDP-based Data Transfer Protocol

UID User IDentification

UML Unified Modeling Language

UML User-Mode Linux

venet Virtual NETwork

veth Virtual ETHernet

VIOLIN Virtual Internetworking on OverLay INfrastructure

VL Virtual Link

VM Virtual Machine

VMM Virtual Machine Monitor

VMTP Versatile Message Transaction Protocol

VNC Virtual Network Computing

VNE Virtual Network Engine

VoIP Voice over IP

VPN Virtual Private Network

VPS Virtual Private Server

XenBEE Xen-Based Execution Environment

Bibliography

[1] D. J. Abadi, Y. Ahmad, M. Balazinska, U. Cetintemel, M. Cherniack, J.-H. Hwang, W. Lindner, A. S. Maskey, A. Rasin, E. Ryvkina, N. Tatbul, Y. Xing, and S. B. Zdonik: "The Design of the Borealis Stream Processing Engine." In *Proceedings of the Third Biennial Conference on Innovative Data Systems Research (CIDR'05)*, Asilomar, CA, USA, 2005.

[2] L. Abeni, T. Cucinotta, G. Lipari, L. Marzario, and L. Palopoli: "QoS Management Through Adaptive Reservations," Real-Time Systems, 29(2-3):131–155, 2005, ISSN 0922-6443.

[3] D. Addison, J. Beecroft, D. Hewson, M. McLaren, D. Roweth, and D. Kidger: *QsNet^{II}: Performance Evaluation*, May 2004. Available online (February 2009): `http://www.quadrics.com/Quadrics/QuadricsHome.nsf/DisplayPages/27F079793A86F93C802575460051C2ED/$File/Performance.pdf`.

[4] B. Agarwalla, N. Ahmed, D. Hilley, and U. Ramachandran: "Streamline: A Scheduling Heuristic for Streaming Application on the Grid." In *Thirteenth Annual Multimedia Computing and Networking Conference (MMCN'06)*, San Jose, CA, Jan 2006.

[5] I. F. Akyildiz, W. Su, Y. Sankarasubramaniam, and E. Cayirci: "A survey on sensor networks," Communications Magazine, IEEE, 40(8):102–114, 2002.

[6] D. S. Alex, M. W. Hicks, P. Kakkar, A. D. Keromytis, M. Shaw, J. T. Moore, C. A. Gunter, T. Jim, S. M. Nettles, and J. M. Smith: "The switchware active network implementation." In *The 1998 ACM SIGPLAN Workshop on ML held in conjunction with the International Conference on Functional Programming (ICFP) '98*, 1998.

[7] D. S. Alex, P. B. Menage, A. D. Keromytis, W. A. Arbaugh, K. G. Anagnostakis, and J. M. Smith: "The Price of Safety in an Active Network," Communications and Networks (JCN), Special Issue on Programmable Switches and Routers, 3(1):4–18, 2001.

[8] D. S. Alexander, W. A. Arbaugh, A. D. Keromytis, and J. M. Smith: "A Secure Active Network Architecture: Realization in SwitchWare," IEEE Network Special Issue on Active and Controllable Networks, 12(3):37–45, 1998.

[9] D. S. Alexander, B. Braden, C. A. Gunter, A. W. Jackson, A. D. Keromytis, G. J. Minden, and D. Wetherall: *Active Network Encapsulation Protocol (ANEP)*, 1997. RFC draft. Available online (February 2009): `http://www.cis.upenn.edu/~switchware/ANEP/docs/ANEP.txt`.

[10] D. Alexander, W. Arbaugh, M. Hicks, P. Kakkar, A. Keromytis, J. Moore, C. Gunder, S. Nettles, and J. Smith: "The SwitchWare Active Network Architecture," IEEE Network Special Issue on Active and Controllable Networks, 12(3):29–36, 1998.

[11] J. Allen, A. Christie, W. Fithen, J. McHugh, J. Pickel, and E. Stoner: *State of the Practice of Intrusion Detection Technologies*. Technical Report CMU/SEI-99-TR-028, ESC-99-028, Carnegie Mellon University.

[12] G. Ammons, J. Appavoo, M. Butrico, D. Da Silva, D. Grove, K. Kawachiya, O. Krieger, B. Rosenburg, E. Van Hensbergen, and R. W. Wisniewski: "Libra: a library operating system for a jvm in a virtualized execution environment." In *VEE '07: Proceedings of the 3rd international conference on Virtual execution environments*, pages 44–54, New York, NY, USA, 2007. ACM, ISBN 978-1-59593-630-1.

[13] J. J. Amor, G. Robles, J. M. Gonzalez-Barahona, and J. F.-S. Pena: *Measuring Etch: the size of Debian 4.0*, 2007. Available online (June 2009): `https://penta.debconf.org/~joerg/attachments/33-measuring_etch_slides.pdf`.

[14] M. J. Anderson, M. Moffie, and C. I. Dalton: *Towards Trustworthy Virtualisation Environments: Xen Library OS Security Service Infrastructure*. Technical Report HPL-2007-69, Hewlett-Packard Laboratories, April 2007. `http://www.hpl.hp.com/techreports/2007/HPL-2007-69.pdf`.

[15] O. Andreasson: "Iptables Tutorial 1.2.0," Available online (February 2009): `http://www.askarali.org/Iptables%20Tutorial%201.2.0.pdf`, 2005.

[16] S. Andreozzi, S. Burke, F. Ehm, L. Field, G. Galang, B. Konya, M. Litmaath, P. Millar, and J. Navarro: *Glue schema specification, version 2.0*, 2008. Available online (June 2009): `http://forge.gridforum.org/sf/projects/glue-wg/`.

[17] A. Anjomshoaa, F. Brisard, M. Drescher, D. Fellows, A. Ly, S. McGough, D. Pulsipher, and A. Savva: *Job Submission Description Language (JSDL) Specification, Version 1.0*, 2005. Global Grid Forum. Available online (June 2009): `http://forge.gridforum.org/projects/jsdl-wg`.

[18] D. Antoš, J. Denemark, and J. Fousek: *Effects of Virtualisation on Behaviour and Performance Characteristics of Network Processing*. Technical Report 3/2008, CESNET, 2008. `http://www.cesnet.cz/doc/techzpravy/2008/effects-of-virtualisation-on-network-processing/`.

[19] P. Apparao, S. Makineni, and D. Newell: "Characterization of network processing overheads in Xen." In *VTDC '06: Proceedings of the 2nd International Workshop on Virtualization Technology in Distributed Computing*, page 2, Washington, DC, USA, 2006. IEEE Computer Society, ISBN 0-7695-2873-1.

[20] W. A. Arbaugh, D. J. Farber, and J. M. Smith: "A Secure and Reliable Bootstrap Architecture." In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 65–71, Oakland, CA, 1997. IEEE Computer Society.

[21] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. Katz, A. Konwinski, G. Lee, D. Patterson, A. Rabkin, I. Stoica, and M. Zaharia: *Above the Clouds: A Berkeley View of Cloud Computing*. Technical report, University of California at Berkeley, February 2009. `http://berkeleyclouds.blogspot.com/2009/02/above-clouds-released.html`.

[22] A. Baiocchi, A. Castellani, and F. Vacirca: "YeAH-TCP: yet Another Highspeed TCP." In *Fifth International Workshop on Protocols for FAST Long-Distance Networks (PFLDnet-07)*, pages 37–42, February 2007. `http://wil.cs.caltech.edu/pfldnet2007/paper/YeAH\_TCP.pdf`.

[23] J. Barnes: *Programming in Ada 2005*. Addison-Wesley, Harlow, 2006, ISBN 0-321-34078-7.

[24] J. Beecroft, D. Addison, F. Petrini, and M. McLaren: *QsNet^{II}: An Interconnect for Supercomputing Applications*, 2004. Available online (February 2009): `http://hpc.pnl.gov/people/fabrizio/papers/hot03.pdf`.

[25] F. Bellard: "QEMU, a Fast and Portable Dynamic Translator." In *Proceedings of the USENIX 2005 Annual Technical Conference, FREENIX Track*, pages 41–46, 2005. Available online (January 2009): `http://www.usenix.org/events/usenix05/tech/freenix/bellard.html`.

[26] B. N. Bershad, C. Chambers, S. J. Eggers, C. Maeda, D. McNamee, P. Pardyak, S. Savage, and E. G. Sirer: "SPIN - An Extensible Microkernel for Application-specific Operating System Services." In *ACM SIGOPS European Workshop*, pages 68–71, 1994.

[27] B. N. Bershad, S. Savage, E. G. Sirer, M. Fiuczynski, D. Becker, S. Eggers, and C. Chambers: "Extensibility, safety and performance in the SPIN operating system." In *Proceedings of the 15th ACM Symposium on Operating System Principles (SOSP-15)*, pages 267–284. Copper Mountain, 1995.

[28] S. Bhatia, M. Motiwala, W. Muhlbauer, Y. Mundada, V. Valancius, A. Bavier, N. Feamster, L. Peterson, and J. Rexford: "Trellis: A platform for building flexible, fast virtual networks on commodity hardware." In *ACM CoNEXT 2008, Real Overlays & Distributed Systems (ROADS'08) workshop*, Madrid, Spain, December 2008.

[29] S. Bhatia, M. Motiwala, W. Muhlbauer, V. Valancius, A. Bavier, N. Feamster, L. Peterson, and J. Rexford: *Hosting Virtual Networks on Commodity Hardware*. Technical report, Georgia Technical University, Georgia, November 2008.

[30] B. Blunden: *Virtual machine design and implementation in C/C++*. Wordware Publishing, Plano, TX, USA, 2002, ISBN 1-55622-903-8.

[31] J. Blythe, E. Deelman, A. Gil, C. Kesselman, A. Agarwal, G. Mehta, and K. Vahi: "The role of planning in grid computing." In *Proceedings of the International Conference on Automated Planning and Scheduling (ICAPS)*, pages 9–13, 2003.

[32] N. Boden, D. Cohen, R. Felderman, A. Kulawik, C. Seitz, J. Seizovic, and W.-K. Su: "Myrinet: a gigabit-per-second local area network," Micro, IEEE, 15(1):29–36, Feb 1995, ISSN 0272-1732.

[33] G. Booch, J. Rumbaugh, and I. Jacobson: *The Unified Modeling Language User Guide*. Addison Wesley, Reading, Massachusetts, 2nd edition, May 2005, ISBN 0321267974.

[34] G. Boss, P. Malladi, D. Quan, L. Legregni, and H. Hall: *Cloud computing*. High Performance On Demand Solutions. IBM Corporation, 2007. Available online (May 2009): `http://download.boulder.ibm.com/ibmdl/pub/software/dw/wes/hipods/Cloud_computing_wp_final_8Oct.pdf`.

[35] D. Bovet and M. Cesati: *Understanding the Linux Kernel, Second Edition*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, 2002, ISBN 0596002130.

[36] R. Braden, L. Zhang, S. Berson, S. Herzog, and S. Jamin: *Resource ReSerVation Protocol (RSVP)*, 1997. RFC no. 2205. Available online (February 2009): `ftp://ftp.isi.edu/in-notes/rfc2205.txt`.

[37] L. S. Brakmo and L. L. Peterson: "TCP Vegas: end to end congestion avoidance on a global Internet," IEEE Journal on selected Areas in communications, 13:1465–1480, 1995.

[38] M. Branson, F. Douglis, B. Fawcett, Z. Liu, A. Riabov, and F. Ye: "CLASP: collaborating, autonomous stream processing systems." In *Middleware '07: Proceedings of the ACM/IFIP/USENIX 2007 International Conference on Middleware*, pages 348–367, New York, NY, USA, 2007. Springer-Verlag New York, Inc.

[39] T. Braun, M. Günter, M. Kasumi, and I. Khalil: *Virtual Private Network Architecture*, 1999. Deliverable no. CATI-IAM-DE-I-000-1.1. CATI–Charging and Accounting Technology for the Internet. Available online (June 2009): `http://www.iam.unibe.ch/~rvs/research/publications/CATI-IAM-DN-P-000-1.1.pdf`.

[40] P. G. Bridges, G. T. Wong, M. Hiltunen, R. D. Schlichting, and M. J. Barrick: "A configurable and extensible transport protocol," IEEE/ACM Transactions on Networking, 15(6):1254–1265, 2007, ISSN 1063-6692.

[41] L. Brim, I. Černá, P. Vařeková, and B. Zimmerova: "Component-interaction automata as a verification-oriented component-based system specification," SIGSOFT Software Engineering Notes, 31(2):4, 2006, ISSN 0163-5948.

[42] M. A. Brown: *Traffic Control HOWTO*, 2006. Available online (March 2009): `http://linux-ip.net/articles/Traffic-Control-HOWTO/`.

[43] E. Bugnion, S. Devine, K. Govil, and M. Rosenblum: "Disco: running commodity operating systems on scalable multiprocessors," ACM Transactions on Computer Systems, 15(4):412–447, 1997, ISSN 0734-2071.

[44] S. Burke: *Xen Networking*, 2007. Available online (March 2009): `http://wiki.kartbuilding.net/index.php/Xen_Networking`.

[45] M. Butrico, D. Da Silva, O. Krieger, M. Ostrowski, B. Rosenburg, D. Tsafrir, E. Van Hensbergen, R. W. Wisniewski, and J. Xenidis: "Specialized execution environments," SIGOPS Operating Systems Review, 42(1):106–107, 2008, ISSN 0163-5980.

[46] R. Buyya, C. S. Yeo, and S. Venugopal: "Market-oriented Cloud Computing: Vision, Hype, and Reality for Delivering IT Services as Computing Utilities." In *Proceedings of the 10th IEEE International Conference on High Performance Computing and Communications (HPCC 2008)*, pages 5–13, Dalian, China, September 2008. IEEE Computer Society: Los Alamitos, CA, USA.

[47] C. Caini and R. Firrincieli: "TCP Hybla: a TCP enhancement for heterogeneous networks," International Journal of Satellite Communications and Networking, 22(5):547–566, August 2004, ISSN 1542-0981.

[48] K. L. Calvert: *Architectural framework for active networks (version 1.0)*, 1999. Available online (June 2009): `http://protocols.netlab.uky.edu/~calvert/arch-latest.ps`.

[49] A. T. Campbell, H. G. De Meer, M. E. Kounavis, K. Miki, J. B. Vicente, and D. Villela: "A survey of programmable networks," SIGCOMM Computer Communication Review, 29(2):7–23, 1999, ISSN 0146-4833.

[50] A. T. Campbell, M. E. Kounavis, D. A. Villela, J. B. Vicente, I. Corporation, and H. G. D. Meer: "Spawning networks," IEEE Network, 13:16–29, 1999.

[51] A. T. Campbell, H. G. D. Meer, M. E. Kounavis, K. Miki, J. B. Vicente, and D. Villela: "A survey of programmable networks," SIGCOMM Computer Communication Review, 29(2):7–23, 1999, ISSN 0146-4833.

[52] R. Campbell: *Managing AFS: the Andrew File System*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1998, ISBN 0-13-802729-3.

[53] A. Caprara, H. Kellerer, and U. Pferschy: "The Multiple Subset Sum Problem," SIAM Journal on Optimization, 11(2):308–319, 2000, ISSN 1052-6234.

[54] L. Cardelli, J. Donahue, and L. Glassman: *Modula-3 report (revised)*. Available online (February 2009): `http://www.hpl.hp.com/techreports/Compaq-DEC/SRC-RR-52.html`.

[55] R. Cardoe, J. Finney, A. C. Scott, and D. Shepherd: "LARA: A Prototype System for Supporting High Performance Active Networking." In *International Working Conference on Active and Programmable Networks (IWAN'99)*, pages 117–131, 1999.

[56] G. Carter, J. Ts, and R. Eckstein: *Using Samba*. O'Reilly Media, Inc., Sebastopol, CA, USA, January 2007, ISBN 0-596-00769-8.

[57] S. Carter, M. Minich, and N. S. V. Rao: "Experimental evaluation of infiniband transport over local- and wide-area networks." In *SpringSim '07: Proceedings of the 2007 spring simulation multiconference*, pages 419–426, San Diego, CA, USA, 2007. Society for Computer Simulation International, ISBN 1-56555-313-6.

[58] P. M. Chen and B. D. Noble: "When Virtual Is Better Than Real." In *HOTOS '01: Proceedings of the Eighth Workshop on Hot Topics in Operating Systems*, page 133, Washington, DC, USA, 2001. IEEE Computer Society.

[59] D. Cheriton: *VMTP: Versatile Message Transaction Protocol*, 1988. RFC no. 1045. Available online (May 2009): `ftp://ftp.isi.edu/in-notes/rfc1045.txt`.

[60] L. Cherkasova and R. Gardner: "Measuring CPU overhead for I/O processing in the Xen virtual machine monitor." In *ATEC '05: Proceedings of the annual conference on USENIX Annual Technical Conference*, pages 24–24, Berkeley, CA, USA, 2005. USENIX Association.

[61] L. Cherkasova, D. Gupta, and A. Vahdat: "Comparison of the three CPU schedulers in Xen," SIGMETRICS Performormance Evaluation Review, 35(2):42–51, September 2007, ISSN 0163-5999.

[62] B. Chun, D. Culler, T. Roscoe, A. Bavier, L. Peterson, M. Wawrzoniak, and M. Bowman: "PlanetLab: an overlay testbed for broad-coverage services," SIGCOMM Computer Communication Review, 33(3):3–12, July 2003, ISSN 0146-4833.

[63] A. Cichocki, M. Rusinkiewicz, and D. Woelk: *Workflow and Process Automation: Concepts and Technology*. Kluwer Academic Publishers, Norwell, MA, USA, 1998, ISBN 0792380991.

[64] C. Clark, K. Fraser, S. Hand, J. G. Hansen, E. Jul, C. Limpach, I. Pratt, and A. Warfield: "Live Migration of Virtual Machines." In *Proceedings of the 2nd ACM/USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 273–286, Boston, MA, May 2005.

[65] D. D. Clark, M. L. Lambert, and L. Zhang: *NETBLT: A bulk data transfer protocol*, 1987. RFC no. 998. Available online (May 2009): `ftp://ftp.isi.edu/in-notes/rfc998.txt`.

[66] S. H. Clearwater: *Market-based control: a paradigm for distributed resource allocation*. World Scientific Publishing Co., Inc., River Edge, NJ, USA, 1996, ISBN 981-02-2254-8.

[67] B. Coppin: *Artificial Intelligence Illuminated*. Jones and Bartlett Publishers, Inc., USA, 2004, ISBN 0763732303.

[68] D. Cunningham, B. Lane, and W. Lane: *Gigabit Ethernet Networking*. Macmillan Publishing Co., Inc. Indianapolis, IN, USA, 1999.

[69] D. Decasper, G. Parulkar, S. Choi, J. Dehart, T. Wolf, and B. Plattner: "A scalable, high performance active network node," IEEE Network, 13:8–19, 1999.

[70] D. Decasper and B. Plattner: "DAN: Distributed Code Caching for Active Networks." In *IEEE Conference on Computer Communications (INFOCOM'98)*, pages 609–616, 1998.

[71] S. Denazis: *D4-Revised Active Node Architecture and Design*, 2002. Second version of the FAIN Active Router Design, Deliverable no. 4 to the Future Active IP Networks EU project (IST-1999-10561-FAIN). Available online (June 2009): `http://www.ist-fain.org/deliverables/del4/d4.pdf`.

[72] J. Denemark: *Autentizace v aktivních sítích (authentication in active networks)*. Master's thesis, Faculty of Informatics, Masaryk University in Brno, April 2003. Czech only.

[73] J. Denemark, M. Ruda, and L. Matyska: *Virtualizing METACenter Resources Using Magrathea*. Technical Report 25/2007, CESNET, z. s. p. o., November 2007.

[74] J. Dike: "User-mode Linux." In *Proceedings of the USENIX 5th Annual Linux Showcase & Conference*, pages 3–14, 2001. Available online (January 2009): `http://www.usenix.org/publications/library/proceedings/als01/dike.html`.

[75] W. Dong: *Migrating Linux VServers*. Department of Computer Science, Princeton University. Available online (January 2009): `http://www.cs.princeton.edu/~wdong/wiki/uploads/Main/Courses/COS518_final.pdf`.

[76] B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, I. Pratt, A. Warfield, P. B. m, and R. Neugebauer: "Xen and the Art of Virtualization." In *Proceedings of the ACM Symposium on Operating Systems Principles*, Bolton Landing, NY, USA, October 2003.

[77] K. J. Duda and D. R. Cheriton: "Borrowed-virtual-time (BVT) scheduling: supporting latency-sensitive threads in a general-purpose scheduler." In *SOSP '99: Proceedings of the seventeenth ACM symposium on Operating systems principles*, volume 33, pages 261–276, New York, NY, USA, December 1999. ACM Press.

[78] E. Hladká and M. Liška and T. Rebok: "Stereoscopic Video over IP Networks." In *Proceedings of the International Conference on Networking and Services (ICNS'05)*, 2005.

[79] D. Farinacci, T. Li, S. Hanks, D. Meyer, and P. Traina: *Generic Routing Encapsulation (GRE)*, 2000. RFC no. 2784. Available online (March 2009): `ftp://ftp.isi.edu/in-notes/rfc2784.txt`.

[80] D. Ferrari, A. Gupta, and G. Ventre: "Distributed advance reservation of real-time connections," Multimedia Syst., 5(3):187–198, 1997, ISSN 0942-4962.

[81] P. Ferrie: *Attacks on more Virtual Machine Emulators*, 2006. Symantec Advanced Threat Research. Available online (June 2009): `http://www.symantec.com/avcenter/reference/Virtual_Machine_Threats.pdf`.

[82] R. J. Figueiredo, P. A. Dinda, and J. A. B. Fortes: "A Case For Grid Computing On Virtual Machines." In *ICDCS '03: Proceedings of the 23rd International Conference on Distributed Computing Systems*, page 550, Washington, DC, USA, 2003. IEEE Computer Society, ISBN 0-7695-1920-2.

[83] J. Filipovič, I. Peterlík, and L. Matyska: "On-line Precomputation Algorithm for Real-time Haptic Interaction with Non-linear Deformable Bodies." In *Proceedings of The Third Joint EuroHaptics Conference and Symposium on Haptic Interfaces for Virtual Environments and Teleoperator Systems*, pages 24–29, 2009.

[84] D. Flanagan and Y. Matsumoto: *The Ruby Programming Language*. O'Reilly Media, Inc., Upper Saddle River, NJ, USA, January 2008, ISBN 0596516177.

[85] B. Ford, P. Srisuresh, and D. Kegel: "Peer-to-peer communication across network address translators." In *USENIX Annual Technical Conference*, pages 179–192, 2005.

[86] S. Forrest, S. A. Hofmeyr, and A. Somayaji: "A sense of self for unix processes." In *Proceedings of the 1996 IEEE Symposium on Security and Privacy*, pages 120–128. IEEE Computer Society Press, 1996.

[87] I. T. Foster: "Globus Toolkit Version 4: Software for Service-Oriented Systems." In *Proceedings of Network and Parallel Computing conference (NPC'05)*, volume 3779 of *Lecture Notes in Computer Science*, pages 2–13. Springer, 2005.

[88] I. Foster, C. Kesselman, C. Lee, B. Lindell, K. Nahrstedt, and A. Roy: "A Distributed Resource Management Architecture that Supports Advance Reservations and Co-Allocation." In *Proceedings of the International Workshop on Quality of Service*, pages 27–36, 1999.

[89] I. Foster, A. Roy, and V. Sander: "A Quality of Service Architecture that Combines Resource Reservation and Application Adaptation." In *Proceedings of the Eight International Workshop on Quality of Service (IWQOS 2000)*, pages 181–187, 2000.

[90] I. Foster, T. Freeman, K. Keahy, D. Scheftner, B. Sotomayer, and X. Zhang: "Virtual Clusters for Grid Communities." In *CCGRID '06: Proceedings of the Sixth IEEE International Symposium on Cluster Computing and the Grid*, pages 513–520, Washington, DC, USA, 2006. IEEE Computer Society, ISBN 0-7695-2585-7.

[91] K. A. Fraser, S. M. Hand, T. L. Harris, I. M. Leslie, and I. A. Pratt: *The Xenoserver computing infrastructure*. Technical Report UCAM-CL-TR-552, University of Cambridge, Computer Laboratory, January 2003. `http://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-552.pdf`.

[92] M. J. Frisch, G. W. Trucks, H. B. Schlegel, G. E. Scuseria, M. A. Robb, J. R. Cheeseman, J. A. Montgomery, Jr., T. Vreven, K. N. Kudin, J. C. Burant, J. M. Millam, S. S. Iyengar, J. Tomasi, V. Barone, B. Mennucci, M. Cossi, G. Scalmani, N. Rega, G. A. Petersson, H. Nakatsuji, M. Hada, M. Ehara, K. Toyota, R. Fukuda, J. Hasegawa, M. Ishida, T. Nakajima, Y. Honda, O. Kitao, H. Nakai, M. Klene, X. Li, J. E. Knox, H. P. Hratchian, J. B. Cross, V. Bakken, C. Adamo, J. Jaramillo, R. Gomperts, R. E. Stratmann, O. Yazyev, A. J. Austin, R. Cammi, C. Pomelli, J. W. Ochterski, P. Y. Ayala, K. Morokuma, G. A. Voth, P. Salvador, J. J. Dannenberg, V. G. Zakrzewski, S. Dapprich, A. D. Daniels, M. C. Strain, O. Farkas, D. K. Malick, A. D. Rabuck, K. Raghavachari, J. B. Foresman, J. V. Ortiz, Q. Cui, A. G. Baboul, S. Clifford, J. Cioslowski, B. B. Stefanov, G. Liu, A. Liashenko, P. Piskorz, I. Komaromi, R. L. Martin, D. J. Fox, T. Keith, M. A. Al-Laham, C. Y. Peng, A. Nanayakkara, M. Challacombe, P. M. W. Gill, B. Johnson, W. Chen, M. W. Wong, C. Gonzalez, and J. A. Pople: *Gaussian 03, Revision E.01*. Gaussian, Inc., Wallingford, CT, 2004.

[93] X. Fu, W. Shi, A. Akkerman, and V. Karamcheti: "CANS: composable, adaptive network services infrastructure." In *USITS'01: Proceedings of the 3rd conference on USENIX Symposium on Internet Technologies and Systems*, Berkeley, CA, USA, 2001. USENIX Association.

[94] A. Galis, S. Denazis, C. Brou, and C. Klein: *Programmable Networks for IP Service Deployment*. Artech House, Inc., Norwood, MA, USA, 2004, ISBN 1580537456.

[95] A. Galis, B. Plattner, J. M. Smith, S. G. Denazis, E. Moeller, H. Guo, C. Klein, J. Serrat, J. Laarhuis, G. T. Karetsos, and C. Todd: "A Flexible IP Active Networks Architecture." In *International Working Conference on Active and Programmable Networks (IWAN'00)*, pages 1–15, 2000.

[96] M. R. Garey and D. S. Johnson: *Computers and Intractability : A Guide to the Theory of NP-Completeness (Series of Books in the Mathematical Sciences)*. W. H. Freeman & Co Ltd, January 1979, ISBN 0716710455.

[97] T. Garfinkel, B. Pfaff, J. Chow, M. Rosenblum, and D. Boneh: "Terra: a virtual machine-based platform for trusted computing," SIGOPS Operating Systems Review, 37(5):193–206, 2003, ISSN 0163-5980.

[98] T. Garfinkel and M. Rosenblum: "A virtual machine introspection based architecture for intrusion detection." In *Proceedings of Network and Distributed Systems Security Symposium*, pages 191–206, February 2003.

[99] M. Garofalakis, J. Gehrke, and R. Rastogi: *Data Stream Management: Processing High-Speed Data Streams (Data-Centric Systems and Applications)*. Springer-Verlag New York, Inc. Secaucus, NJ, USA, 2007.

[100] B. Ghose, V. Jain, and V. Gopal: *Characterizing QoS-Awareness in Multimedia Operating Systems*. Available online (June 2009): `http://www.cs.ucsd.edu/classes/fa99/cse221/OSSurveyF99/papers/ghose,jain,gopal.characterizing_QoS-awareness_in_multimedia_operating_systems.pdf`, 1999.

[101] C. Gloman and M. J. Pescatore: *Working with HDV: Shoot, Edit, and Deliver Your High Definition Video*. Focal Press, 2006, ISBN 0240808886.

[102] R. P. Goldberg: "Architecture of virtual machines." In *Proceedings of the workshop on Virtual Computer Systems*, pages 74–112, New York, NY, USA, 1973. ACM Press.

[103] O. Goldreich: *Computational Complexity: A Conceptual Perspective*. Cambridge University Press, 1 edition, April 2008, ISBN 052188473X.

[104] M. Golm, M. Felser, C. Wawersich, and J. Kleinöder: "The JX Operating System." In *ATEC '02: Proceedings of the General Track of the annual conference on USENIX Annual Technical Conference*, pages 45–58, Berkeley, CA, USA, 2002. USENIX Association, ISBN 1-880446-00-6.

[105] J. S. Gray: *Interprocess Communications in Linux: The Nooks and Crannies*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2003, ISBN 0-13-046042-7.

[106] S. D. Gribble, M. Welsh, R. v. Behren, E. A. Brewer, D. Culler, N. Borisov, S. Czerwinski, R. Gummadi, J. Hill, A. Joseph, R. H. Katz, Z. M. Mao, S. Ross, B. Zhao, and R. C. Holte: "The Ninja architecture for robust Internet-scale systems and services," Computer Networks, 35(4):473–497, 2001, ISSN 1389-1286.

[107] Y. Gu and R. L. Grossman: "UDT: UDP-based data transfer for high-speed wide area networks," Computer Networks, 51(7):1777–1799, May 2007.

[108] J. Guo, F. Chen, L. Bhuyan, and R. Kumar: "A Cluster-Based Active Router Architecture Supporting Video/Audio Stream Transcoding Service." In *IPDPS '03: Proceedings of the 17th International Symposium on Parallel and Distributed Processing*, page 44.2, Washington, DC, USA, 2003. IEEE Computer Society, ISBN 0-7695-1926-1.

[109] D. Gupta, L. Cherkasova, R. Gardner, and A. Vahdat: "Enforcing Performance Isolation Across Virtual Machines in Xen." In *Proceedings of the 7th ACM/IFIP/USENIX Middleware Conference*. Melbourne, Australia, November 2006.

[110] D. Gupta, R. Gardner, and L. Cherkasova: *XenMon: QoS Monitoring and Performance Profiling tool*. Technical report, HP Labs, No. HPL-2005-187, 2005.

[111] S. Ha, I. Rhee, and L. Xu: "CUBIC: a new TCP-friendly high-speed TCP variant," SIGOPS Operating Systems Review, 42(5):64–74, 2008, ISSN 0163-5980.

[112] W. Hagen: *Professional Xen Virtualization*. Wrox, Indianapolis, 2007, ISBN 0470138114.

[113] S. Hand, T. L. Harris, E. Kotsovinos, and I. Pratt: "Controlling the XenoServer Open Platform." In *Proceedings of the 6th International Conference on Open Architectures and Network Programming (OPENARCH'03)*, April 2003.

[114] S. Haubold, H. Mix, W. E. Nagel, and M. Romberg: "The unicore grid and its options for performance analysis," Performance analysis and grid computing, pages 275–288, 2004.

[115] B. Hayes: "Cloud Computing," Commun. ACM, 51(7):9–11, 2008, ISSN 0001-0782.

[116] J. P. Hespanha, M. Mclaughlin, G. S. Sukhatme, M. Akbarian, R. Garg, and W. Zhu: "Haptic Collaboration over the Internet." In *Proceedings of the Fifth Phantom Users Group Workshop*, pages 9–13. Prentice Hall, 2000.

[117] M. Hicks, P. Kakkar, J. T. Moore, C. A. Gunter, and S. Nettles: "PLAN: A Packet Language for Active Networks." In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 86–93, 1998.

[118] M. Hicks, J. T. Moore, D. S. Alex, C. A. Gunter, and S. M. Nettles: "PLANet: an Active Internetwork." In *Proceedings of the Eighteenth IEEE Computer and Communication Society INFOCOM Conference*, pages 1124–1133. IEEE, 1999.

[119] M. Hicks, A. Nagarajan, and R. van Renesse: *User-specified Adaptive Scheduling in a Streaming Media Network*. Technical Report CS-TR-4430, University of Maryland Department of Computer Science, March 2003.

[120] E. Hladká: *User Empowered Collaborative Environment: Active Network Support*. PhD thesis, Faculty of Informatics, Masaryk University in Brno, 2004.

[121] E. Hladká, P. Holub, and J. Denemark: "User Empowered Virtual Multicast for Multimedia Communication." In *3rd International Conference on Networking (ICN'04)*, pages 338–343, Gosier, Guadeloupe, March 2004.

[122] E. Hladká and Z. Salvet: "An Active Network Architecture: Distributed Computer or Transport Medium." In P. Lorenz (editor): *Networking – ICN 2001: First International Conference Colmar, France, July 9-13, 2001, Proceedings, Part II*, volume 2094 of *Lecture Notes in Computer Science*, pages 612–619, Heidelberg, January 2001. Springer-Verlag.

[123] T. Hoefler, T. Mehlan, A. Lumsdaine, and W. Rehm: "Netgauge: A Network Performance Measurement Framework." In *High Performance Computing and Communications, Third International Conference, HPCC 2007, Houston, USA, September 26-28, 2007, Proceedings*, volume 4782, pages 659–671. Springer, Sep. 2007, ISBN 978-3-540-75443-5.

[124] D. Hoffman, D. Prabhakar, and P. Strooper: "Testing iptables." In *CASCON '03: Proceedings of the 2003 conference of the Centre for Advanced Studies on Collaborative research*, pages 80–91. IBM Press, 2003.

[125] S. A. Hofmeyr, S. Forrest, and A. Somayaji: "Intrusion detection using sequences of system calls," Journal of Computer Security, 6(3):151–180, 1998, ISSN 0926-227X.

[126] P. Holub: *Network and Grid Support for Multimedia Distribution and Processing*. PhD thesis, Faculty of Informatics, Masaryk University in Brno, May 2005.

[127] P. Holub, E. Hladká, J. Denemark, and T. Rebok: "Active Elements for High-Definition Video Distribution." In *ICT 2006, 13th International Conference on Telecommunications*, pages 1–4, Funchal, Madeira: University of Aveiro, Portugal, March 2006.

[128] R. Hughes-Jones, P. Clarke, and S. Dallison: "Performance of 1 and 10 Gigabit Ethernet cards with server quality motherboards," Future Generation Computer Systems, 21(4):469–488, 2005, ISSN 0167-739X.

[129] G. Humphreys, M. Houston, R. Ng, R. Frank, S. Ahern, P. D. Kirchner, and J. T. Klosowski: "Chromium: a stream-processing framework for interactive rendering on clusters." In *SIGGRAPH Asia '08: ACM SIGGRAPH ASIA 2008 courses*, pages 1–10, New York, NY, USA, 2008. ACM.

[130] J. G. Hurwitz and W.-c. Feng: "End-to-End Performance of 10-Gigabit Ethernet on Commodity Systems," IEEE Micro, 24(1):10–22, 2004, ISSN 0272-1732.

[131] Information Sciences Institute, University of Southern California: *Transmission Control Protocol*, 1981. RFC no. 793. Available online (February 2009): `ftp://ftp.isi.edu/in-notes/rfc793.txt`.

[132] A.-A. Ivan, J. Harman, M. Allen, and V. Karamcheti: "Partitionable Services: A Framework for Seamlessly Adapting Distributed Applications to Heterogeneous Environments." In *HPDC '02: Proceedings of the 11th IEEE International Symposium on High Performance Distributed Computing*, page 103, Washington, DC, USA, 2002. IEEE Computer Society, ISBN 0-7695-1686-6.

[133] V. Jacobson: "Congestion avoidance and control," SIGCOMM Computer Communication Review, 18(4):314–329, August 1988, ISSN 0146-4833.

[134] T. Jaeger: *Operating Systems Security*. Morgan & Claypool Publishers, San Rafael, CA, USA, 2007, ISBN 1598292129.

[135] X. Jiang and D. Xu: "VIOLIN: Virtual Internetworking on Overlay Infrastructure." In J. Cao, L. T. Yang, M. Guo, and F. Lau (editors): *Parallel and Distributed Processing and Applications: Second International Symposium*. Springer Berlin / Heidelberg, December 2004.

[136] M. B. Jones, P. J. Leach, R. P. Draves, and J. S. Barrera: "Modular real-time resource management in the Rialto operating system." In *HOTOS '95: Proceedings of the Fifth Workshop on Hot Topics in Operating Systems (HotOS-V)*, page 12, Washington, DC, USA, 1995. IEEE Computer Society, ISBN 0-8186-7081-9.

[137] M. T. Jones: *Cloud computing with Linux*. Developer Works. IBM Corporation, 2009. Available online (May 2009): `http://download.boulder.ibm.com/ibmdl/pub/software/dw/linux/l-cloud-computing/l-cloud-computing-pdf.pdf`.

[138] M. Jordan: "JavaGuest-A Research Java Virtual Machine on Xen." In *Xen Summit*, 2007.

[139] P. Kakkar, M. Hicks, J. T. Moore, and C. A. Gunter: "Specifying the PLAN networking programming language." In *Higher Order Operational Techniques in Semantics*, volume 26 of *Electronic Notes in Theoretical Computer Science*. Elsevier, September 1999.

[140] P. H. Kamp and R. N. M. Watson: "Jails: Confining the omnipotent root." In *Proceedings of the 2nd International SANE Conference*, 2000.

[141] V. Kashyap: *IP over InfiniBand (IPoIB) Architecture*, 2006. RFC no. 4392. Available online (February 2009): `ftp://ftp.isi.edu/in-notes/rfc4392.txt`.

[142] D. Katz: *IP Router Alert Option*, 1997. RFC no. 2113. Available online (February 2009): `http://www.ietf.org/rfc/rfc2113.txt`.

[143] K. Keahey, K. Doering, and I. Foster: "From Sandbox to Playground: Dynamic Virtual Environments in the Grid." In *GRID '04: Proceedings of the 5th IEEE/ACM International Workshop on Grid Computing*, pages 34–42, Washington, DC, USA, 2004. IEEE Computer Society, ISBN 0-7695-2256-4.

[144] K. Keahey, I. Foster, T. Freeman, and X. Zhang: "Virtual workspaces: Achieving quality of service and quality of life in the Grid," Scientific Programming, 13(4):265–275, 2005, ISSN 1058-9244.

[145] N. Kelem and R. Feiertag: "A Separation Model for Virtual Machine Monitors." In *Research in Security and Privacy: IEEE Computer Society Symposium*, pages 78–86, 1991.

[146] R. Keller, S. Choi, M. Dasen, D. Decasper, G. Fankhauser, and B. Plattner: "An active router architecture for multicast video distribution." In *INFOCOM 2000: Nineteenth Annual Joint Conference of the IEEE Computer and Communications Societies*, pages 1137–1146, 2000.

[147] R. Keller, L. Ruf, A. Guindehi, and B. Plattner: "PromethOS: A Dynamically Extensible Router Architecture Supporting Explicit Routing." In *IWAN '02: Proceedings of the IFIP-TC6 4th International Working Conference on Active Networks*, pages 20–31, London, UK, 2002. Springer-Verlag, ISBN 3-540-00223-5.

[148] S. Kent and K. Seo: *Security Architecture for the Internet Protocol*, 2005. RFC no. 4301. Available online (February 2009): `ftp://ftp.isi.edu/in-notes/rfc4301.txt`.

[149] C. Kesselman and I. Foster: *The Grid: Blueprint for a New Computing Infrastructure*. Morgan Kaufmann Publishers, November 1998, ISBN 1558604758.

[150] G. C. Kessler: *An Overview of Cryptography*. Auerbach Publications, 1998. Available online (June 2009): `http://www.garykessler.net/library/crypto.html`.

[151] T. Kichkaylo: *Construction of Component-Based Applications by Planning*. PhD thesis, Department of Computer Science, New York University, 2005.

[152] T. Kichkaylo and V. Karamcheti: "Optimal Resource-Aware Deployment Planning for Component-Based Distributed Applications." In *HPDC '04: Proceedings of the 13th IEEE International Symposium on High Performance Distributed Computing*, pages 150–159, Washington, DC, USA, 2004. IEEE Computer Society, ISBN 0-7803-2175-4.

[153] A. Kivity, Y. Kamay, D. Laor, U. Lublin, and A. Liguory: "KVM: the Linux Virtual Machine Monitor." In *Proceedings of the Linux Symposium*, pages 225–230, 2007.

[154] N. Kiyanclar, G. A. Koenig, and W. Yurcik: "Maestro-VC: A Paravirtualized Execution Environment for Secure On-Demand Cluster Computing." In *CCGRID '06: Proceedings of the Sixth IEEE International Symposium on Cluster Computing and the Grid*, page 28, Washington, DC, USA, 2006. IEEE Computer Society, ISBN 0-7695-2585-7.

[155] E. Kohler: *The Click Modular Router*. PhD thesis, Massachusetts Institute of Technology, USA, 2001.

[156] E. Kohler, R. Morris, B. Chen, J. Jannotti, and F. M. Kaashoek: "The click modular router," ACM Transactions on Computer Systems, 18(3):263–297, August 2000, ISSN 0734-2071.

[157] E. Kohler, M. Handley, and S. Floyd: *Datagram Congestion Control Protocol (DCCP)*, 2006. RFC no. 4340. Available online (February 2009): `ftp://ftp.isi.edu/in-notes/rfc4340.txt`.

[158] E. Kotsovinos and T. Harris: "Distributed resource discovery and management in XenoServers." In *7th CaberNet Radicals Workshop*. Bertinoro, Italy, October 2002.

[159] M. E. Kounavis: *Programming Network Architectures*. PhD thesis, Graduate School of Arts and Sciences, Columbia University, 2004.

[160] M. E. Kounavis, A. T. Campbell, S. Chou, F. Modoux, J. Vicente, and H. Zhuang: "The Genesis Kernel: A Programming System for Spawning Network Architectures," IEEE Journal on Selected Areas in Communications, 19:511–526, 2001.

[161] A. Krall: "Efficient JavaVM just-in-time compilation." In *International Conference on Parallel Architectures and Compilation Techniques*, pages 205–212, 1998.

[162] O. Krieger, M. Auslander, B. Rosenburg, R. W. Wisniewski, J. Xenidis, D. Da Silva, M. Ostrowski, J. Appavoo, M. Butrico, M. Mergen, A. Waterland, and V. Uhlig: "K42: building a complete operating system." In *EuroSys '06: Proceedings of the 2006 EuroSys conference*, pages 133–145, New York, NY, USA, 2006. ACM Press, ISBN 1595933220.

[163] I. Krsul, A. Ganguly, J. Zhang, J. A. B. Fortes, and R. J. Figueiredo: "VMPlants: Providing and Managing Virtual Machine Execution Environments for Grid Computing." In *SC '04: Proceedings of the 2004 ACM/IEEE conference on Supercomputing*, page 7, Washington, DC, USA, 2004. IEEE Computer Society, ISBN 0-7695-2153-3.

[164] P. J. M. Laarhoven and E. H. L. Aarts (editors): *Simulated Annealing: theory and applications*. Kluwer Academic Publishers, Norwell, MA, USA, 1987.

[165] C. Lattner and V. Adve: "The LLVM Compiler Framework and Infrastructure Tutorial." In *LCPC'04 Mini Workshop on Compiler Research Infrastructures*, West Lafayette, Indiana, Sep 2004.

[166] M. Laureano, C. Maziero, and E. Jamhour: "Protecting host-based intrusion detectors through virtual machines," Computer Networks, 51(5):1275–1283, 2007, ISSN 1389-1286.

[167] K. Lawton: *Bochs: The open source IA-32 emulation project*. Available online (Februrary 2009): `http://bochs.sourceforge.net/`.

[168] A. A. Lazar, K.-S. Lim, and F. Marconcini: "Realizing a foundation for programmability of ATM networks with the binding architecture," IEEE Journal of Selected Areas in Communications, 14(7):1214–1227, 1996.

[169] A. A. Lazar, A. Temple, A. Temple, R. Gidron, and R. Gidron: "An Architecture for Integrated Networks that Guarantees Quality of Service," International Journal of Digital and Analog Communication Systems, 3:229–238, 1990.

[170] L.-W. H. Lehman, S. J. Garland, and D. L. Tennenhouse: "Active Reliable Multicast." In *INFOCOM*, pages 581–589, 1998.

[171] X. Leroy, D. Doligez, J. Garrigue, D. Rémy, and J. Vouillon: *The Objective Caml System release 3.09: Documentation and user's manual*, 2006. Available online (June 2009): `http://caml.inria.fr/`.

[172] I. M. Leslie, D. Mcauley, R. Black, T. Roscoe, P. T. Barham, D. Evers, R. Fairbairns, and E. Hyden: "The Design and Implementation of an Operating System to Support Distributed Multimedia Applications," IEEE Journal of Selected Areas in Communications, 14(7):1280–1297, 1996.

[173] A. Leung and L. George: "Static single assignment form for machine code." In *PLDI '99: Proceedings of the ACM SIGPLAN 1999 conference on Programming language design and implementation*, pages 204–214, New York, NY, USA, 1999. ACM Press.

[174] S. Levi, S. K. Tripathi, S. D. Carson, and A. K. Agrawala: "The MARUTI hard real-time operating system," SIGOPS Operating Systems Review, 23(3):90–105, 1989, ISSN 0163-5980.

[175] H. Levkowetz and S. Vaarala: *Mobile IP Traversal of Network Address Translation (NAT) Devices*, May 2003. RFC no. 3519. Available online (February 2009): `ftp://ftp.isi.edu/in-notes/rfc3519.txt`.

[176] B. des Ligneris: "Virtualization of Linux Based Computers: The Linux-VServer Project." In *HPCS '05: Proceedings of the 19th International Symposium on High Performance Computing Systems and Applications*, pages 340–346, Washington, DC, USA, 2005. IEEE Computer Society, ISBN 0-7695-2343-9.

[177] A. Liguori: *QEMU Emulator User Documentation*, 2009. Available online (January 2009): `http://www.nongnu.org/qemu/qemu-doc.html`.

[178] T. Lindholm and F. Yellin: *The Java(TM) Virtual Machine Specification (2nd Edition)*. Prentice Hall PTR, April 1999, ISBN 0201432943.

[179] J. Liu, W. Huang, B. Abali, and D. K. Panda: "High performance VMM-bypass I/O in virtual machines." In *ATEC '06: Proceedings of the annual conference on USENIX '06 Annual Technical Conference*, page 14, Berkeley, CA, USA, 2006. USENIX Association.

[180] I. Llorente: *OpenNebula Project*. Available online (May 2009): `http://www.opennebula.org/`.

[181] S. E. Madnick and J. J. Donovan: "Application and analysis of the virtual machine approach to information system security and isolation." In *Proceedings of the workshop on virtual computer systems*, pages 210–224, New York, NY, USA, 1973. ACM.

[182] V. Maraia: *The Build Master*. Addison-Wesley Professional, Toronto, Ontario, Canada, 2005.

[183] S. Martello and P. Toth: *Knapsack Problems: Algorithms and Computer Implementations*. John Wiley & Sons Inc, 1990.

[184] T. Martínek and M. Košek: "NetCOPE: Platform for Rapid Development of Network Applications." In *Proc. of 2008 IEEE Design and Diagnostics of Electronic Circuits and Systems Workshop*, pages 219–224. IEEE Computer Society, 2008, ISBN 978-1-4244-2276-0. `http://www.fit.vutbr.cz/research/view_pub.php?id=8607`.

[185] H. Massalin: *Synthesis: An Efficient Implementation of Fundamental Operating System Services*. PhD thesis, Graduate School of Arts and Sciences, Columbia University, 1992.

[186] C. McCain: *Mastering VMware Infrastructure 3 (Mastering)*. SYBEX Inc., Alameda, CA, USA, 2008, ISBN 0470183136, 9780470183137.

[187] R. Mcillroy: *Network Router Resource Virtualisation*. Master's thesis, University of Glasgow, 2005.

[188] R. McIlroy and J. Sventek: "Resource Virtualisation of Network Routers." In *International Workshop on High Performance Switching and Routing (HPSR)*. IEEE, 2006.

[189] M. K. McKusick and G. V. Neville-Neil: *The design and implementation of the FreeBSD operating system*. Pearson Education, 2004.

[190] H. Mel: *Cryptography Decrypted*. Addison-Wesley, Boston, 2001, ISBN 0201616475.

[191] A. Menon, A. L. Cox, and W. Zwaenepoel: "Optimizing network virtualization in Xen." In *ATEC '06: Proceedings of the annual conference on USENIX '06 Annual Technical Conference*, pages 2–2, Berkeley, CA, USA, 2006. USENIX Association.

[192] A. Menon, J. R. Santos, Y. Turner, G. J. Janakiraman, and W. Zwaenepoel: "Diagnosing performance overheads in the Xen virtual machine environment." In *VEE '05: Proceedings of the 1st ACM/USENIX international conference on Virtual execution environments*, pages 13–23, New York, NY, USA, 2005. ACM, ISBN 1-59593-047-7.

[193] J. E. V. D. Merwe, S. Rooney, I. M. Leslie, and S. A. Crosby: "The Tempest - A Practical Framework for Network Programmability," IEEE Network, 12:20–28, 1997.

[194] A. Mirkin, A. Kuznetsov, and K. Kolyshkin: "Containers checkpointing and live migration." In *Proceedings of the Linux Symposium*, pages 85–90, 2008.

[195] J. Mo, R. J. La, V. Anantharam, and J. Walr: "Analysis and comparison of TCP Reno and Vegas." In *Proceedings of IEEE Infocom*, pages 1556–1563, 1999.

[196] A. Montz, D. Mosberger, S. O'Mally, L. Peterson, and T. Proebsting: "Scout: a communications-oriented operating system," Hot Topics in Operating Systems, Workshop on, 0:58, 1995.

[197] T. Moreton: *A wide-area file system for migrating virtual machines*. Technical Report UCAM-CL-TR-714, University of Cambridge, Computer Laboratory, March 2008. `http://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-714.pdf`.

[198] D. Mosberger and L. L. Peterson: "Making paths explicit in the Scout operating system," SIGOPS Operating Systems Review, 30(SI):153–167, 1996, ISSN 0163-5980.

[199] E. Mouw: "Linux kernel procfs guide," Relatório técnico, Delfty University of Technology an Systems, 2001.

[200] R. Müller, G. Alonso, and D. Kossmann: "A virtual machine for sensor networks," SIGOPS Operating Systems Review, 41(3):145–158, 2007, ISSN 0163-5980.

[201] R. Müller, G. Alonso, and D. Kossmann: "SwissQM: Next Generation Data Processing in Sensor Networks." In *Proceedings of the Third Biennial Conference on Innovative Data Systems Research (CIDR'03)*, pages 1–9. Asilomar, CA, USA, January 2007.

[202] K. Nahrstedt, H.-h. Chu, and S. Narayan: "QoS-aware resource management for distributed multimedia applications," Journal of High Speed Networks, 7(3-4):229–257, 1998, ISSN 0926-6801.

[203] K. Naveen, V. Venkatram, C. Vaidya, S. Nicholas, S. Allan, Z. Charles, G. Gideon, L. Jason, and J. Andrew: "SAGE: the Scalable Adaptive Graphics Environment." In *Proceedings of the WACE 2004 Conference*. Nice, France, 2004.

[204] E. L. Nygren, S. J. Garl, and M. F. Kaashoek: "PAN: A high-performance active network node supporting multiple mobile code systems." In *Proceedings of the International Conference on Open Architectures and Network Programming (OpenArch'99)*, pages 78–89, 1999.

[205] E. Nygren: *The design and implementation of a high-performance active network node*. Master's thesis, Massachusetts Institute of Technology, February 1998.

[206] T. Ormandy: *An Empirical Study into the Security Exposure to Hosts of Hostile Virtualized Environments*, 2007. Available online (June 2009): `http://taviso.decsystem.org/virtsec.pdf`.

[207] M. Ott, G. Welling, S. Mathur, D. Reininger, and R. Izmailov: "The JOURNEY active network model," IEEE Journal on Selected Areas in Communications, 19(3), 2001.

[208] G. Parulkar, D. C. Schmidt, and J. S. Turner: "aitpm: a strategy for integrating ip with atm," SIGCOMM Computer Communication Review, 25(4):49–59, 1995, ISSN 0146-4833.

[209] S. Paul, K. K. Sabnani, J. C. Lin, and S. Bhattacharyya: "Reliable Multicast Transport Protocol (RMTP)," IEEE Journal on Selected Areas in Communications, 15(3):1414–1424, 1997.

[210] I. Peterlík and L. Matyska: "An Algorithm of State-Space Precomputation Allowing Non-linear Haptic Deformation Modelling Using Finite Element Method." In *WHC '07: Proceedings of the Second Joint EuroHaptics Conference and Symposium on Haptic Interfaces for Virtual Environment and Teleoperator Systems*, pages 231–236, Washington, DC, USA, 2007. IEEE Computer Society, ISBN 0-7695-2738-8.

[211] L. Peterson, Y. Gottlieb, M. Hibler, P. Tullmann, J. Lepreau, S. Schwab, H. Dandelkar, A. Purtell, and J. Hartman: "An OS Interface for Active Routers," IEEE Journal on Selected Areas in Communications, 19(3):473–487, March 2001.

[212] L. Peterson, S. Muir, T. Roscoe, and A. Klingaman: *PlanetLab Architecture: An Overview*. Technical Report PDN–06–031, PlanetLab Consortium, May 2006.

[213] L. Peterson and T. Roscoe: "The design principles of PlanetLab," SIGOPS Operating Systems Review, 40(1):11–16, 2006, ISSN 0163-5980.

[214] F. Petrini, W. chun Feng, A. Hoisie, S. Coll, and E. Frachtenberg: "The Quadrics Network: High-Performance Clustering Technology," IEEE Micro, 22(1):46–57, 2002, ISSN 0272-1732.

[215] A. Petry: *Design and Implementation of a Xen-Based Execution Environment*. Master's thesis, The University of Kaiserslautern, Department of Computer Science, Kaiserslautern, Germany, April 2007.

[216] T. Peuker: *An Object-Oriented Architecture for the Real-Time Transmission of Multimedia Data Streams*. Master's thesis, Institut für Mathematische Maschinen und Datenverarbeitung (Informatik), University of Erlangen-Nürnberg, 1997.

[217] H. Pillay: *Setting up IP Aliasing on A Linux Machine MinižHOWTO*, 2001. Available online (April 2009): `http://www.puppozungo.com/documenti/IP-Aliasing-HowTo-EN.pdf`.

[218] D. Pisinger: "An exact algorithm for large multiple knapsack problems," European Journal of Operational Research, 114:528–541, 1999.

[219] B. Podey, T. Kessler, and H.-D. Melzer: "Network Packet Filter Design and Performance." In *Lecture Notes in Computer Science – Information Networking*, volume 2662, pages 803–816. Springer, 2003.

[220] J. Postel: *User Datagram Protocol*, 1980. RFC no. 768. Available online (February 2009): `ftp://ftp.isi.edu/in-notes/rfc768.txt`.

[221] J. Postel: *Internet Protocol (IP)*, 1981. RFC no. 791. Available online (February 2009): `ftp://ftp.isi.edu/in-notes/rfc791.txt`.

[222] K. Psounis: *Active Networks: Applications, Security, Safety, and Architectures*, 1999. IEEE Communications Surveys, 2(1):2–16, 1999.

[223] C. Pu, T. Autrey, A. Black, C. Consel, C. Cowan, J. Inouye, L. Kethana, J. Walpole, and K. Zhang: "Optimistic incremental specialization: streamlining a commercial operating system," SIGOPS Operating Systems Review, 29(5):314–321, 1995, ISSN 0163-5980.

[224] K. R. Rao, Z. S. Bojkovic, and D. A. Milovanovic: *Multimedia Communication Systems: Techniques, Standards, and Networks*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2002, ISBN 013031398X.

[225] T. Rebok: *Active Router Communication Layer*. Technical Report 11/2004, CESNET, 2004. `http://www.cesnet.cz/doc/techzpravy/2004/artp-protocol/`.

[226] T. Rebok: *Měření plánovacích algoritmů prototypu aktivního směrovače s podporou QoS*. Available online (March 2009): `http://www.fi.muni.cz/~xrebok/DiProNN_QoS/DiProNN_QoS.pdf`, 2008.

[227] R. van Renesse, K. P. Birman, R. Friedman, M. Hayden, and D. A. Karr: "A framework for protocol composition in Horus." In *PODC '95: Proceedings of the fourteenth annual ACM symposium on Principles of distributed computing*, pages 80–89, New York, NY, USA, 1995. ACM, ISBN 0-89791-710-3.

[228] D. A. Reynolds: "An overview of automatic speaker recognition technology." In *Proceedings of International Conference on Acoustics, Speech, and Signal Processing (ICASSP'02)*, pages 4072–4075, vol.4, May 2002.

[229] A. Riabov and Z. Liu: "Scalable Planning for Distributed Stream Processing Systems." In *Proceedings of the Sixteenth International Conference on Automated Planning and Scheduling, ICAPS 2006*, pages 31–41, Cumbria, UK, 2006.

[230] E. Rosen, A. Viswanathan, and R. Callon: *Multiprotocol Label Switching Architecture*, January 2001. RFC no. 3031. Proposed Standard. Available online (February 2009): `ftp://ftp.isi.edu/in-notes/rfc3031.txt`.

[231] M. J. Rutten, J. T. V. Eijndhoven, E.-J. D. Pol, E. G. Jaspers, P. V. der Wolf, O. P. Gangwal, and A. Timmer: "Eclipse: Heterogeneous Multiprocessor Architecture for Flexible Media Processing," International Parallel and Distributed Processing Symposium, 2:0130b, 2002.

[232] F. Sacerdoti, M. Katz, M. Massie, and D. Culler: "Wide Area Cluster Monitoring with Ganglia," Proceedings of the IEEE Cluster 2003 Conference, 2003.

[233] S. Schmid: *A component-based active router architecture*. PhD thesis, Lancaster University, UK, 2002.

[234] S. Schmid: *LARA++ Design Specification*, 2000. Lancaster University DMRG Internal Report, MPG-00-03, January 2000.

[235] S. Schmid, J. Finney, A. Scott, and W. Shepherd: "Component-based Active Networks for Mobile Multimedia Systems." In *Network and Operating Systems Support for Digital Audio and Video (NOSSDAV'2000), Chapel Hill, North Carolina, June 2000*, 2000.

[236] H. Schulzrinne, S. Casner, R. Frederick, and V. Jacobson: *RTP: A Transport Protocol for Real-Time Applications*, 2003. RFC no. 3550. Available online (February 2009): `ftp://ftp.isi.edu/in-notes/rfc3550.txt`.

[237] B. D. Schuymer: *Ebtables Hacking HOWTO*, 2003. Available online (March 2009): `http://ebtables.sourceforge.net/ebtables-hacking/ebtables-hacking-HOWTO.html`.

[238] B. D. Schuymer and N. Fedchik: *ebtables/iptables interaction on a Linux-based bridge*, 2003. Available online (March 2009): `http://ebtables.sourceforge.net/br_fw_ia/br_fw_ia.html`.

[239] P. Schwan: "Lustre: Building a File System for 1,000-node Clusters." In *Proceedings of the 2003 Linux Symposium*, July 2003.

[240] B. Schwartz, A. W. Jackson, T. W. Strayer, W. Zhou, D. R. Rockwell, and C. Partridge: "Smart Packets for active networks." In *Proceedings of the International Conference on Open Architectures and Network Programming (OPENARCH'99)*, pages 90–97, 1999.

[241] B. Schwartz, A. W. Jackson, W. T. Strayer, W. Zhou, R. D. Rockwell, and C. Partridge: "Smart packets: applying active networks to network management," ACM Transactions on Computer Systems, 18:21, 2000.

[242] J. Seward, N. Nethercote, J. Weidendorfer, and the Valgrind Development Team: *Valgrind 3.3 – Advanced Debugging and Profiling for GNU/Linux applications*. Network Theory Ltd., Bristol, UK, March 2008.

[243] J. S. Shapiro and N. Hardy: "EROS: A Principle-Driven Operating System from the Ground Up," IEEE Software, 19(1):26–33, 2002, ISSN 0740-7459.

[244] S. Shepler, B. Callaghan, D. Robinson, R. Thurlow, C. Beame, M. Eisler, and D. Noveck: *Network File System (NFS) version 4 Protocol*, 2003. RFC no. 3530. Available online (March 2009): `ftp://ftp.isi.edu/in-notes/rfc3530.txt`.

[245] S. D. Silva, D. Florissi, and Y. Yemini: "Composing Active Services in Netscript." In *DARPA Active Networks Workshop*. Tucson, AZ, 1998.

[246] M. Sipser: "The history and status of the P versus NP question." In *STOC '92: Proceedings of the twenty-fourth annual ACM symposium on Theory of computing*, pages 603–618, New York, NY, USA, 1992. ACM, ISBN 0-89791-511-9.

[247] J. E. Smith and R. Nair: *Virtual Machines: Versatile Platforms for Systems and Processes*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2005, ISBN 1558609105.

[248] O. Spatscheck and L. L. Peterson: "Defending Against Denial of Service Attacks in Scout." In *Proceedings of the USENIX/ACM Symposium on Operating System Design and Implementation*, pages 59–72, 1999.

[249] N. Spivack: *Welcome to the Stream: The Next Phase of the Web*, May 2009. Available online (June 2009): `http://www.twine.com/item/128lryv9z-46/welcome-to-the-stream-the-next-phase-of-the-web`.

[250] W. Stallings: *Cryptography and network security: principles and practice*. Prentice Hall, 2006, ISBN 0131873164.

[251] W. R. Stevens: *UNIX network programming, volume 2 (2nd ed.): interprocess communications*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1999, ISBN 0-13-081081-9.

[252] H. Sturgis, J. Mitchell, and J. Israel: "Issues in the design and use of a distributed file system," SIGOPS Operating Systems Review, 14(3):55–69, 1980, ISSN 0163-5980.

[253] V. Sundaram, A. Chandra, P. Goyal, P. Shenoy, J. Sahni, and H. Vin: "Application performance in the QLinux multimedia operating system." In *MULTIMEDIA '00: Proceedings of the eighth ACM international conference on Multimedia*, pages 127–136, New York, NY, USA, 2000. ACM, ISBN 1-58113-198-4.

[254] SWsoft: *OpenVZ User's Guide*, 2005. Available online (February 2009): `http://download.openvz.org/doc/OpenVZ-Users-Guide.pdf`.

[255] C. Szyperski: *Component Software: Beyond Object-Oriented Programming*. ACM Press and Addison-Wesley, New York, NY, 1998.

[256] V. Talwar, B. Agarwalla, S. Basu, R. Kumar, and K. Nahrstedt: "A resource allocation architecture with support for interactive sessions in utility Grids." In *CCGRID '04: Proceedings of the 2004 IEEE International Symposium on Cluster Computing and the Grid*, pages 731–734, Washington, DC, USA, 2004. IEEE Computer Society, ISBN 0-7803-8430-X.

[257] A. S. Tanenbaum and A. S. Woodhull: *Operating Systems Design and Implementation (3rd Edition) (Prentice Hall Software Series)*. Prentice Hall, January 2006, ISBN 0131429388.

[258] D. Tarditi, S. Puri, and J. Oglesby: *Accelerator: simplified programming of graphics processing units for general-purpose uses via data parallelism*. Technical report, Microsoft Research, Microsoft Corporation, 2005.

[259] D. Thain, T. Tannenbaum, and M. Livny: *Condor and the grid*. In *Grid Computing: Making the Global Infrastructure a Reality*. John Wiley & Sons Inc., December 2002, ISBN 978-0-470-85319-1.

[260] S. Thibault and T. Deegan: "Improving performance by embedding HPC applications in lightweight Xen domains." In *HPCVirt '08: Proceedings of the 2nd workshop on System-level virtualization for high performance computing*, pages 9–15, New York, NY, USA, 2008. ACM, ISBN 978-1-60558-120-0.

[261] W. Thies, M. Karczmarek, and S. Amarasinghe: "StreamIt: A Language for Streaming Applications." In *International Conference on Compiler Construction*, Grenoble, France, April 2002. `http://cag.lcs.mit.edu/commit/papers/02/streamit-cc.pdf`.

[262] L. Tian: *Resource Allocation in Streaming Environments*. Master's thesis, California Institute of Technology, Pasadena, California, May 2006.

[263] H. Tokuda, T. Nakajima, and P. Rao: "Real-time Mach: Towards a predictable real-time system." In *Proceedings of USENIX Mach Workshop*, pages 73–82, 1990.

[264] P. Tullmann, M. Hibler, and J. Lepreau: "Janos: A Java-oriented OS for active network nodes," IEEE Journal on Selected Areas in Communications, Active and Programmable Networks, 2001.

[265] E. Van Hensbergen: "The effect of virtualization on OS interference." In *Proceedings of the 1st Annual Workshop on Operating System Interference in High Performance Applications*, August 2005.

[266] E. Van Hensbergen: "P.R.O.S.E.: partitioned reliable operating system environment," SIGOPS Operating Systems Review, 40(2):12–15, April 2006, ISSN 0163-5980.

[267] E. Walker: "A distributed file system for a wide-area high performance computing infrastructure." In *WORLDS'06: Proceedings of the 3rd conference on USENIX Workshop on Real, Large Distributed Systems*, pages 9–9, Berkeley, CA, USA, 2006. USENIX Association.

[268] B. Ward: *The Book of VMware: The Complete Guide to VMware Workstation*. No Starch Press, Inc., San Francisco, CA, 2002, ISBN 1886411727.

[269] J. van Waveren: *Real-Time DXT Compression*, 2006. Id Software, Inc. Available online (June 2009): `http://cache-www.intel.com/cd/00/00/32/43/324337_324337.pdf`.

[270] D. X. Wei, C. Jin, S. H. Low, and S. Hegde: "FAST TCP: motivation, architecture, algorithms, performance," IEEE/ACM Transactions on Networking, 14(6):1246–1259, 2006, ISSN 1063-6692.

[271] S. A. Weil, S. A. Brandt, E. L. Miller, D. D. E. Long, and C. Maltzahn: "Ceph: a scalable, high-performance distributed file system." In *OSDI '06: Proceedings of the 7th symposium on Operating systems design and implementation*, pages 307–320, Berkeley, CA, USA, 2006. USENIX Association, ISBN 1931971471.

[272] R. Weinreich and J. Sametinger: *Component-based software engineering: putting the pieces together*. Addison-Wesley Longman Publishing Co., Inc., 2001, ISBN 978-0201704853.

[273] A. Weiss: "Computing in the Clouds," netWorker, 11(4):16–25, 2007, ISSN 1091-3556.

[274] G. Welling, M. Ott, and S. Mathur: "A Cluster-Based Active Router Architecture," IEEE Micro, 21(1):16–25, 2001, ISSN 0272-1732.

[275] M. Welzl and M. Mühlhäuser: "Scalability and Quality of Service: A Trade-off?," IEEE Communications Magazine, pages 32–36, June 2003.

[276] D. Wetherall, J. Guttag, and D. Tennenhouse: "ANTS: Network Services Without the Red Tape," Computer, 32(4):42–48, 1999, ISSN 0018-9162.

[277] D. Wetherall, J. Guttag, and D. Tennenhouse: *ANTS: A Toolkit for Building and Dynamically Deploying Network Protocols*, 1998. Proceedings of the International Conference on Open Architectures and Network Programming (OPENARCH'98).

[278] A. Whitaker, M. Shaw, and S. D. Gribble: "Denali: Lightweight Virtual Machines for Distributed and Networked Applications." In *Proceedings of the USENIX Annual Technical Conference*, 2002.

[279] A. Whitaker, M. Shaw, and S. D. Gribble: "Scale and performance in the Denali isolation kernel," SIGOPS Operating Systems Review, 36(SI):195–209, 2002, ISSN 0163-5980.

[280] M. Willebeek-LeMair, D. D. Kandlur, and Z.-Y. Shae: "On Multipoint Control Units for Viodeoconferencing." In *IEEE Conference on Local Computer Networks (LCN'94)*, pages 356–364, 1994.

[281] P. Willmann, J. Shafer, D. Carr, A. Menon, S. Rixner, A. L. Cox, and W. Zwaenepoel: "Concurrent Direct Network Access for Virtual Machine Monitors." In *High Performance Computer Architecture (HPCA 2007)*, pages 306–317, 2007.

[282] L. C. Wolf, C. Griwodz, and R. Steinmetz: "Multimedia Communication." In *Proceedings of the IEEE*, volume 85, pages 1915–1933, 1997.

[283] L. C. Wolf and R. Steinmetz: "Concepts for Resource Reservation in Advance," Multimedia Tools Appl., 4(3):255–278, 1997, ISSN 1380-7501.

[284] T. Wolf and D. Decasper: "CPU scheduling for active processing using feedback deficit round robin." In *Proceedings of Allerton Conference on Communication, Control, and Computing*, 1999.

[285] C. Xiong, J. Leigh, E. He, V. Vishwanath, T. Murata, L. Renambot, and T. DeFanti: "LambdaStream-a data transport protocol for streaming network-intensive applications over photonic networks." In *Proceedings of The Third International Workshop on Protocols for Fast Long-Distance Networks, Lyon, France*, 2005.

[286] M. D. Yarvis, P. Reiher, and G. Popek: *Conductor: Distributed Adaptation for Heterogeneous Networks*. Kluwer Academic Publishers, Norwell, MA, USA, 2002, ISBN 140207087X.

[287] X. Zheng, A. Mudambi, and M. Veeraraghavan: "FRTP: fixed rate transport protocol – a modified version of SABUL for end-to-end circuits." In *Proceedings of Broadnet'04*. San Jose, CA, 2004.

[288] H. Zimmermann: "OSI Reference Model–The ISO Model of Architecture for Open Systems Interconnection," IEEE Transactions on Communications, 28(4):425–432, 1980.

[289] B. Zimmerova: "Component Placement in Distributed Environment w.r.t. Component Interaction." In *Proceedings of the Doctoral Workshop on Mathematical and Engineering Methods in Computer Science (MEMICS'06)*, pages 260–267. FIT VUT Brno, Czech Republic, October 2006.

[290] *Amazon Elastic Compute Cloud (Amazon EC2)*. Available online (May 2009): `http://aws.amazon.com/ec2/`.

[291] *Ethernet Emulation (TCP/IP and UDP/IP) Performance for GM-2*, 2006. Myricom International. Available online (February 2009): `http://www.myri.com/scs/performance/Myrinet-2000/GM/ip-gm2.html`.

[292] *FreeBSD Handbook*. The FreeBSD Documentation Project. 2009. Available online (April 2009): `http://www.freebsd.org/doc/en/books/handbook/index.html`.

[293] *GM Performance Measurements*, 2006. Myricom International. Available online (February 2009): `http://www.myri.com/scs/performance/Myrinet-2000/GM/`.

[294] *GNU General Public License, version 2*. Available online (January 2009): `http://www.gnu.org/licenses/gpl-2.0.html`.

[295] *Google App Engine*. Available online (May 2009): `http://appengine.google.com/`.

[296] *Guide to Myrinet-2000 Switches and Switch Networks*, August 2001. Myricom International. Available online (June 2009): `http://www.myri.com/myrinet/m3switch/guide/myrinet-2000_switch_guide.pdf`.

[297] *InfiniBand™ Architecture Specification Volume 1 (Release 1.2.1)*, 2007. InfiniBand℠ Trade Association. Available online (February 2009): `http://www.infinibandta.org/specs`.

[298] *Linux VServer Documentation*. Available online (January 2009): `http://linux-vserver.org/Documentation`.

[299] *Linux Containers – Network Namespace*. Available online (June 2009): `http://lxc.sourceforge.net/network.php`.

[300] *Understanding LiquidVM*, 2008. BEA Systems Inc. Available online (March 2009): `http://e-docs.bea.com/wls-ve/docs92-v11/config/lvmintro.html`.

[301] *Microsoft Live Mesh*. Available online (May 2009): `http://www.mesh.com/`.

[302] *Resource Management Guide: ESX Server 3.0.1 and VirtualCenter 2.0.1*, 2009. Available online (April 2009): `http://www.vmware.com/pdf/vi3_301_201_resource_mgmt.pdf`.

[303] *Sun Network.com*. Available online (May 2009): `http://www.network.com/`.

[304] *System S – Stream Computing at IBM Research*, 2008. IBM Research Whitepaper. Available online (June 2009): `http://download.boulder.ibm.com/ibmdl/pub/software/data/sw-library/ii/whitepaper/SystemS_2008-1001.pdf`.

[305] *TG201: 10gigabit ethernet data center switch*, 2008. Product brochure. Available online (February 2009): `http://www.quadrics.com/Quadrics/QuadricsHome.nsf/DisplayPages/30CB00218CB90534802572830004DF81C/$File/TG_201_Feb08.pdf`.

[306] *VIMSH for VMware ESX 3.5*, 2008. Xtravirt limited. Available online (March 2009): `http://knowledge.xtravirt.com/white-papers/scripting.html`.

[307] *VMware ESX Server 2: Administration Guide*, 2006. Available online (April 2009): `http://www.vmware.com/pdf/esx25_admin.pdf`.

[308] *Introduction to VMware Infrastructure*, 2007. Available online (April 2009): `http://www.vmware.com/pdf/vi3_35/esx_3/r35/vi3_35_25_intro_vi.pdf`.

[309] *VMware Scripting API: User's Manual*, 2005. Available online (April 2009): `http://www.vmware.com/pdf/Scripting_API_215.pdf`.

[310] *VMware Server User's Guide (VMware Server version 2.0)*, 2008. Available online (April 2009): `http://www.vmware.com/pdf/vmserver2.pdf`.

[311] *Workstation Userżs Manual*, 2009. Available online (April 2009): `http://www.vmware.com/pdf/ws65_manual.pdf`.

[312] *Xen 3.0 Users' Manual*. Available online (January 2009): `http://tx.downloads.xensource.com/downloads/docs/user/`.

Author's Selected Publications

Journal Papers

[1] Petr Holub, Luděk Matyska, Miloš Liška, Lukáš Hejtmánek, Jiří Denemark, Tomáš Rebok, Andrei Hutanu, Ravi Paruchuri, Jan Radil, and Eva Hladká. High-definition multimedia for multiparty low-latency interactive communication. *Future Generation Computer Systems, Elsevier Science*, 22(8), 2006.

International Conference Papers

[2] Tomáš Rebok. DiProNN Resource Management System. In *Proceedings of Doctoral Workshop on Mathematical and Engineering Methods in Computer Science (MEMICS'08)*, pages 224–231, Znojmo, Czech Republic, 2008.

[3] Tomáš Rebok. DiProNN: Distributed Programmable Network Node Architecture. In *ICNS '08: Proceedings of the Fourth International Conference on Networking and Services*, pages 67–72, Washington, DC, USA, 2008. IEEE Computer Society.

[4] Tomáš Rebok. DiProNN: Distributed Programmable Network Node Architecture. In *Proceedings of Cracow Grid Workshop (CGW'07)*, pages 283–290, Cracow, Poland, 2008. ACC CYFRONET AGH.

[5] Tomáš Rebok. DiProNN Programming Model. In *Proceedings of Doctoral Workshop on Mathematical and Engineering Methods in Computer Science (MEMICS'07)*, pages 177–184, Znojmo, Czech Republic, 2007.

[6] Tomáš Rebok. VM-based Distributed Active Router Design. In *Proceedings of European Computing Conference (ECC'07)*, pages 265–274, Athens, Greece, 2007. LNEE Springer.

[7] Petr Holub, Eva Hladká, Jiří Denemark, and Tomáš Rebok. Active Elements for High-Definition Video Distribution. In *Proceedings of* 13^{th} *International Conference on Telecommunications (ICT'06)*, pages 1–4, Funchal, Madeira, Portugal, March 2006. University of Aveiro.

[8] Tomáš Rebok, Petr Holub, and Eva Hladká. Quality of Service Oriented Active Routers Design. In *Proceedings of MIPRO 2006 / Hypermedia and Grid Systems*, pages 206–211, Opatija, Croatia, May 2006. Croatian Society for Information and Communication Technology, Electronics and Microelectronics.

[9] Tomáš Rebok. VM-based Distributed Active Router Design. In *Proceedings of Doctoral Workshop on Mathematical and Engineering Methods in Computer Science (MEMICS'06)*, pages 190–197, Mikulov, Czech Republic, 2006.

[10] Eva Hladká, Miloš Liška, and Tomáš Rebok. Stereoscopic Video over IP Networks. In *Proceedings of the International Conference on Networking and Services (ICNS'05)*, pages 1–6, Papeete, Tahiti, 2005. IEEE Computer Society.

National Papers

[11] Tomáš Rebok and Petr Holub. Distributed Synchronous Infrastructure for Multimedia Streams Transmission and Processing. In *Networking Studies II : Selected Technical Reports*, pages 147–169, Prague, Czech Republic, 2008. CESNET z. s. p. o.

[12] Tomáš Rebok. DiProNN: Distribuovaný programovatelný síťový prvek s podporou virtuálních strojů. In *Proceedings of Širokopásmové sítě a jejich aplikace 2007*, pages 95–99, Olomouc, Czech Republic, 2007. CESNET, z. s. p. o. and Palacký University.

[13] Tomáš Rebok. Protokoly transportní vrstvy a jejich kategorizace, transportní protokol ARTP. In *Proceedings of Širokopásmové sítě a jejich aplikace 2007*, pages 153–161, Olomouc, Czech Republic, 2005. CESNET, z. s. p. o. and Palacký University.

[14] Michal Procházka, Tomáš Rebok, and Petr Holub. Implementace P2P síti zrcadel v prostředí JXTA. In *Proceedings of Širokopásmové sítě a jejich aplikace 2007*, pages 144–152, Olomouc, Czech Republic, 2005. CESNET, z. s. p. o. and Palacký University.

[15] Eva Hladká, Petr Holub, and Tomáš Rebok. Komunikace s technologií AccesGrid Point. In *Proceedings of Širokopásmové sítě a jejich aplikace 2007*, pages 65–69, Olomouc, Czech Republic, 2003. CESNET, z. s. p. o. and Palacký University.

Non-Reviewed Papers

[16] Tomáš Rebok. Active Router Communication Layer. Technical Report 11/2004, CESNET z. s. p. o., Prague, Czech Republic, 2004.

[17] Tomáš Rebok and Petr Holub. Synchronizing RTP Packet Reflector. Technical Report 7/2003, CESNET z. s. p. o., Prague, Czech Republic, 2003.

Other Presentations

[18] Petr Holub, Ravi Paruchuri, S. Simmons, Tomáš Rebok, Andrei Hutanu, Daniel Eiland, and Miloš Liška. Enabling Technologies for Teaching an HPC Class as a Distributed Course, 2007. Invited talk. Atlanta, Internet2, USA, SURA/ViDe, 2007. SPR2007 - The 9^{th} Annual SURA/ViDe Conference. March 30, 2007.

[19] Petr Holub, Tomáš Rebok, and Miloš Liška. Videokonferenční technologie a přenosy videa, 2005. Invited talk. EurOpen 2005, Šumava, Czech Republic. October 23, 2005.

[20] Tomáš Rebok. DiProNN: VM-based Distributed Programmable Network Node Architecture, 2007. TERENA Networking Conference (TNC'07), Conference poster.

[21] Tomáš Rebok. DiProNN: Distributed Programmable Network Node Architecture, 2007. Cracow Grid Workshop (CGW'07), Conference poster.