DiProNN: Distributed Programmable Network Node Architecture

Tomáš Rebok
Faculty of Informatics
Masaryk University
Brno, Czech Republic
Email: xrebok@fi.muni.cz

***Abstract*—The programmable network approach allows processing of passing user data in a network, which is highly suitable especially for multimedia streams processing. However, programming of complex stream processing applications for programmable nodes is not effortless since they usually do not provide sufficient flexibility (both programming flexibility and execution environment flexibility). In this paper we present the programmable network node architecture named DiProNN that is able to accept and run user-supplied programs and/or virtual machines and process them over passing data. All the DiProNN programs are described using our modular programming model based on the workflow principles that takes advantages of DiProNN virtualization and makes programming of complex streaming applications easier. As a possible application we show a sketch implementation of simple MCU (Multipoint Control Unit) used for large videoconferences that profits from DiProNN properties.**

I. Introduction

The principle called "Active Networks" or "Programmable Networks" is an attempt how to build an intelligent and flexible network using current networks serving as a communication underlay. Such a network allows user-defined processing of passing user data on inner network (programmable) nodes. Multimedia application processing (e.g., video delivery and/or transcoding) or security services (data encryption over untrusted links, secure and reliable multicast, etc.) are a few of possible services which could be provided.

The programmable networks principles became very popular and have been researched by lots of research teams. Various architectures of programmable nodes have been proposed, from integrated ones based on (active) packets containing program code (capsules) to discrete ones (program injection is separated from processing of data packets) based on e.g. software-only or software-hardware architectures. However, programming of complex stream processing applications for programmable nodes is not effortless since they usually do not provide sufficient flexibility (both programming flexibility and execution environment flexibility). Usually, when a program doing 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 new one has to be created.

By employing virtual machines (VMs) principles [1] the flexibility of programmable nodes' execution environment can be increased a lot since they are able to run completely different execution environments simultaneously—the common ones based on common operating systems (e.g. Linux or FreeBSD) as well as the special ones uploaded by users. Moreover, VMs can also bring other benefits—strong isolation, resource management, programming flexibility, etc.—and thus make the usage of programmable nodes easier.

The main goal of our work is to propose the programmable network node architecture named DiProNN (Distributed Programmable Network Node) that is able to accept and run user-supplied programs and/or virtual machines and process them over passing user data. All the DiProNN programs are described using novel modular programming model described in Section III. The model is based on the workflow principles and takes advantages of DiProNN virtualization, thanks to which the programming of complex stream processing applications in DiProNN is more comfortable. As an example application we show an implementation of simple MCU (Multipoint Control Unit) used for large videoconferences that profits from DiProNN properties.

II. DiProNN: Distributed Programmable Network Node

A. Architecture

DiProNN architecture we propose assumes the infrastructure as shown in Figure 1. The DiProNN units form a computer cluster interconnected with each unit having two connections:

- one *low-latency control connection* used for internal communication inside the DiProNN, and
- one *data connection* used for receiving and sending data.

The low latency interconnection is desirable since current common network interfaces like Gigabit Ethernet or 10 Gigabit Ethernet provide large bandwidth, but the latency of the transmission is still in order of hundreds of μs, which is not suitable for fast synchronization of DiProNN units. Thus, the use of specialized low-latency interconnects like Myrinet network providing as low latency as $10\,\mu$s (and even less, if you consider e.g., InfiniBand with $4\,\mu$s), which is close to message passing between threads on a single computer, is very suitable (however, the usage of single interconnection serving as data and control interconnection simultaneously is also possible).

From the high-level perspective of operation, the incoming data are first received by the DiProNN's Distribution unit,

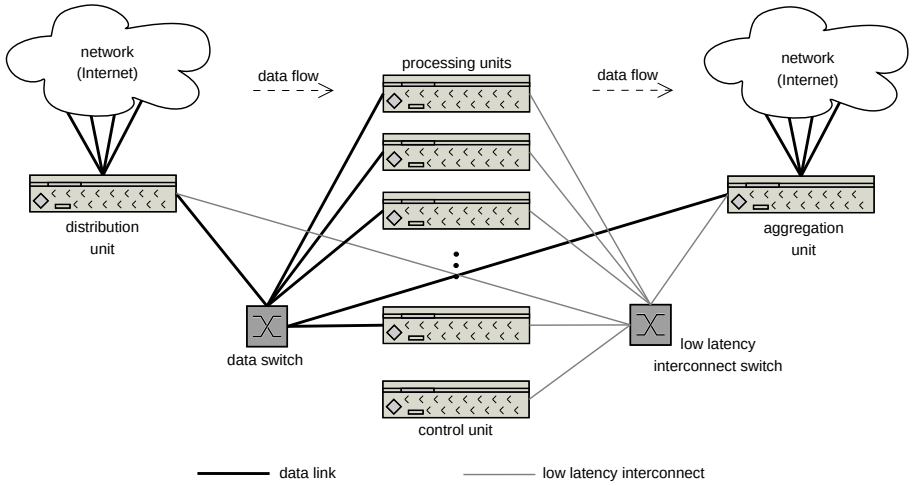


Fig. 1. Proposed DiProNN architecture.

where they are forwarded to appropriate Processing unit(s) for processing. After the whole processing, they 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 1, the DiProNN architecture comprises four major parts:

- **Distribution unit**—the Distribution unit takes care of ingress data flow distribution to appropriate DiProNN Processing unit(s), which are determined by the Control unit described later.
- **Processing units**—the Processing unit (described in detail in Section II-B receives packets and forwards them to proper active programs for processing. The processed data are then forwarded to next active programs for further processing or to the Aggregation unit to be sent away.
  Each Processing unit is also able to communicate with the other ones using the low-latency interconnection. Besides the load balancing and fail over purposes this interconnection is mainly used for sending control information of DiProNN sessions (e.g., state sharing, synchronization, processing control).
- **Control unit**—the Control unit is responsible for the whole DiProNN management and communication with its neighborhood including communication with DiProNN users to negotiate new DiProNN sessions (details about DiProNN sessions establishment are given in Section III) and, if requested, providing feedback about their behavior.
- **Aggregation unit**—the Aggregation unit aggregates the resulting traffic to the output network line(s).

B. DiProNN Processing Units

1) DiProNN and Virtual Machines: The usage of virtual machines enhance the execution environment flexibility of the DiProNN node—they enable DiProNN users not only to upload active programs, which run inside some virtual machine, but they are also allowed to upload a whole virtual machine with its operating system and let their passing data being processed by their own set of active programs running inside uploaded VM(s). Similarly, the DiProNN administrator is able to run his own set of fixed virtual machines, each one with different operating system, and generally with completely different functionality. Furthermore, the VM approach also allows strong isolation among virtual machines, and thus allows strict scheduling of resources to individual VMs, e.g., CPU, memory, and storage subsystem access.

Nevertheless, the VMs also bring some performance overhead necessary for their management [2]. This overhead is especially visible for I/O virtualization, where the Virtual Machine Monitor (VMM) or a privileged host OS has to intervene every I/O operation. We are aware of this performance issues, but we decided to propose a VM-based programmable network node architecture not being limited by current performance restrictions.

2) Processing Unit Architecture: The architecture of the DiProNN Processing unit is shown in Figure 2. The privileged service domain (dom0 in the picture) has to manage the whole Processing unit functionality including uploading, starting and destroying of the virtual machines, communication with the Control unit, and a session accounting and management.

The virtual machines managed by the session management module could be either fixed, providing functionality given by a system administrator, or user-loadable. The example of the fixed virtual machine could be a virtual machine providing classical routing as shown in Figure 2. Besides that, the set of another fixed virtual machines could be started as an active program execution environment where the active programs uploaded by users are executed (those not having their own virtual machine defined). This approach does not force users to upload the whole virtual machine in the case where active program uploading is sufficient.

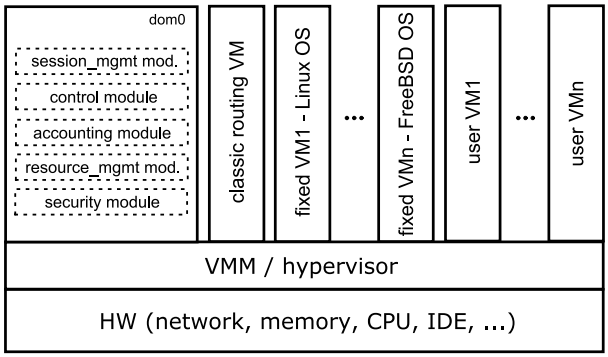


Fig. 2. DiProNN Processing Unit Architecture

C. Communication Protocol

For data transmission, the DiProNN users may use both the *User Datagram Protocol* (UDP) and the transmission protocol called *Active Router Transmission Protocol* (ARTP, [3]) we originally designed and implemented for the generic active router architecture described in [4]. Depending on an applications' demands the users choose the transmission protocol they want to use—whether they want or have to use ARTP's extended functionality (the ARTP is in fact an extension of the UDP protocol like e.g. *Real-time Transport Protocol* (RTP) is) or not. Since the UDP is well-known and widely-used transmission protocol, in the rest of this section we briefly depict the main properties of our ARTP protocol.

1) Active Router Transmission Protocol (ARTP): The ARTP is a connection oriented transport protocol providing reliable duplex communication channel without ensuring that the data will be received in the same order as they were sent. There are two types of data that can be sent by the ARTP protocol:

- *control data* dedicated to both end-point application management and cannot be fragmented,
- *main data* used for the communication itself.

In ARTP, data is transferred using data blocks called ARTP datagrams. The datagrams may have arbitrary size so 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 fragments of a datagram it reassembles them into original datagram and passes the datagram to the receiver application. The order of passing assembled datagrams is not given—the ARTP guarantees the correct datagrams assembling only.

As obvious, the ARTP protocol combines desirable properties of both widely used transport protocols UDP and TCP[1]. On the one hand it provides reliable congestion controlled transport of ARTP datagrams, which are fragmented and thus may have arbitrary size. And on the other, the original sequence of ARTP datagrams is not guaranteed because for the programmable nodes' purposes, the data might be required to be processed in disordered bulks independent on each other.

[1]The ARTP protocol could be used similarly to the UDP transport protocol when having *ARTP datagram = ARTP packet* (not fragmenting at all) and could be also used "almost" similarly to the TCP transport protocol when having ARTP datagram of an "infinite" size.

III. DiProNN Programming Model

In this section we depict a programming model we propose for DiProNN programming. The model is based on the workflow principles [5] and was inspired by the idea of the StreamIt [6], which is a language and compiler specifically designed for modern stream applications programming.

For the DiProNN programming model we adopted the idea of independent simple processing blocks (so called *Filters* in StreamIt), that composed into a processing graph constitute required complex processing. In our case, the processing block is an active program and the communication among such active programs is thanks to the virtualization mechanisms provided by machine hypervisor using common network services (details about DiProNN internal communication are provided in Section IV). The interconnected active programs then compose the *"DiProNN session"* described by its *"DiProNN session graph"*, which is a graphical representation of an *"DiProNN program"* (an example is given in the Figure 3). Furthermore, to make DiProNN programming easier all the active programs as well as the input/output data/communication interfaces are referred by their hierarchical names as shown in the MCU example in Section V.

The DiProNN program defines active programs optionally with virtual machines they run in[2], which are necessary for DiProNN session processing, and defines both data and control communication among them. Besides that, the DiProNN program may also define other parameters (e.g., resources required) of active programs as well as the parameters for the whole DiProNN session.

The main benefit of the DiProNN programming model being described is, that the complex functionality required to be done on the programmable node can be separated into several single-purpose active programs with the data flow among them defined. Furthermore, the usage of symbolic names doesn't force active programs to be aware of their neighbourhood—the active programs processing given DiProNN session before and after them—they are completely independent on each other so that they just have to know the symbolic names of ports they want to communicate with and register them (as sketched in the next section) at the control module of the Processing unit they run in.

A. Data Channels Description

As mentioned in previous section, the DiProNN program defines active programs required for DiProNN session processing, and data and control communication among them. All the input/output data/control interfaces are referred by their hierarchical names and thus each active program has to have these interfaces defined using following structure:

[2]In DiProNN, each active program may run in completely distinct execution environment (e.g., different OS) from the others. However, it is also possible that single VM may contain several active programs running inside.

1) Inputs: For active program inputs definition there must be one parameter named `inputs` in active program parameters section having following structure:

```
inputs =
    in1_name[(DIPRONN_INPUT[(port1)])],
    in2_name[(DIPRONN_INPUT[(port2)])],
    ...
```

where

- the `inX_name` specifies the name of the input port, and
- the `portX` is optional parameter specifying the port number of the DiProNN node input requested.

If the input described is of the form `DIPRONN_INPUT(port)`, it means that the input named `in_name` is the DiProNN node input and the user requests DiProNN to listen on the `port` specified. If the port number is missing, the user indicates that he or she has no preferences on the port they want to make DiProNN listening on (in this situation the real port number where the user should send data to will be negotiated during DiProNN session establishment). The examples of input specifications are given in Figure 3.

2) Outputs: For active program outputs definition and data/control interconnection there must be one parameter named `outputs` in active program parameters section having following structure:

```
outputs =
out1_name(in1 |
   DIPRONN_OUTPUT(receiver1 | SEE_ARTP)),
out2_name(in2 |
   DIPRONN_OUTPUT(receiver2 | SEE_ARTP)),
...
```

where

- the `outX_name` specifies the name of the output port, and
- the `inX` specifies the input port of the next active program where the active program from the port named `outX_name` has to send data to.

If the output described is of the form `DIPRONN_OUTPUT(receiver | SEE_ARTP)`, it means that the output named `out_name` is the DiProNN node output. The DiProNN data receiver is either specified using IP address or domain name, or is defined as an ARTP protocol option of each ARTP packet. The example of an output specification is also given in Figure 3.

B. Internal Control Communication

Sometimes two or more active programs processing given data stream have to communicate with each other (to indicate some new event occurred in e.g. a stream processed in parallel, to share information about their state, or just to ask them for some information). In such cases, the DiProNN users may define control links among active programs similarly as they do it for data communication using keywords `control_inputs` and `control_outputs`. However, in this case the special forms `DIPRONN_INPUT(...)` and `DIPRONN_OUTPUT(...)` have no meaning.

In comparison with data messages, the control messages are not sent via data links, but via an internal low-latency interconnection described in Section II[3] since the latency of control messages transmitted should be as low as possible. Thus, when a port of an active program is registered as a control port, all the messages coming from it to another active program are transmitted using internal low-latency interconnection.

C. Parallel Processing

If the special attribute of an active program

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

is set, the DiProNN performs its processing in parallel. The number of parallel instances running can be either fixed (the requested `instances_count` is present and correctly specified), or variable (controlled by the Control unit depending on actual active program usage and resources available).

Nevertheless, when processing an active program in parallel, the DiProNN session has to specify how to distribute incoming data over such parallel instances. Thus, before every active program having parallelizable attribute set there must be an active program (built-in for e.g. round-robin function, or user-loaded) specifying data distribution over all the parallel instances.

Regarding the control communication, 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 sending control messages from parallelizable AP, the communication channel between both APs is defined in the same way as the one that define control communication between non-parallelizable APs.
2) *parallelizable AP is a receiver of control messages:* if an active program wants to send control message to a parallelizable AP, the communication channel is defined in the same way as the one that define control communication between non-parallelizable APs, and the control messages are broadcasted to all the parallel instances of destined AP.
3) *control communication among parallel instances:* in the case when parallel instances want to communicate with each other, the combination of previous two points take place: in given parallelizable AP there must be an output defined, that is connected to an input of the same parallelizable AP (see an example in Figure 3). In this case, all the control messages coming from specified output are broadcasted to given input of all the parallel instances of given parallelizable AP.

IV. Session Establishment and Data Flow

When a new DiProNN session request arrives to the node, the Distribution unit immediately forwards it to the

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

Control unit. In the situation when the receiver(s) of given DiProNN session is/are known, the Control unit contacts all the DiProNN nodes operating on the path from it to the receiver(s), and asks them for their actual usage. Using the information about their usage the Control unit decides, whether the new DiProNN session request could be satisfied by the first node alone or whether a part of requested DiProNN session has to be (or should be because of resource optimalization) performed on another DiProNN node being on the path from the first DiProNN node to the receiver(s).

If the request could be satisfied, the session establishment takes place. It means, that each DiProNN node receives its relevant part of the whole DiProNN session (including all the active programs and virtual machines images) and the Control unit of each DiProNN node decides, which Processing units each active program/virtual machine will run on. After that, both the control modules (a part of each Processing unit) and the Distribution units of all the DiProNN nodes used are set appropriately. Then all the active programs and virtual machines are started, and moreover, all the requested resources are reserved, if any.

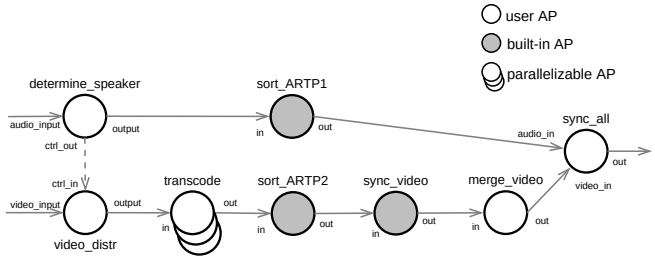
Since the DiProNN programming model uses symbolic names for communication channels (both data and control channels) instead of port numbers, the names must be associated with appropriate port numbers during a DiProNN session startup. This association is done using the control module where each active program using simple text protocol registers the couple (symbolic name, real port). The control module using the information about registered couples together with the virtual machine and port number a packet is coming from properly sets the receiver of passing packets, which are then automatically forwarded to proper active programs.

However, this approach does not enable active programs to know the real data receiver (each packet is by VMM destined to given VM address and given active program's port). Nevertheless, the DiProNN users may use the ARTP's extended functionality to make their active programs being aware of real data receiver as mentioned in Section III-A2. In this case, the Aggregation unit forwards these packets to the destination given inside ARTP datagram instead of the one given in DiProNN program.

V. Example: Simple MCU Unit in DiProNN

In this section we sketch a possible implementation of simple MCU unit (Multipoint Control Unit, [7]) used for videoconferencing. The MCU unit we want to implement should have following functionality: the unit should be able to accept up to twelve input ARTP streams of audio and video data (e.g., 6 audio streams + 6 video streams) of videoconference participants (in this simple case we do not deal with permissions of participants to attend the conference). All the input video streams should be merged into one outgoing video stream and the current speaker should be somehow highlighted (greater picture and/or lighter-colored). The outgoing audio and merged video streams should be synchronized with defined precision.

The possible DiProNN session graph together with a fragment of the DiProNN program is depicted in the Figure 3. First, from incoming audio stream there is a current speaker determined[4] using the `determine_speaker` active program (AP). The identification of current speaker is then using low-latency control interconnection sent to the `video_distr` AP, where relevant ARTP option indicating current speaker is added to the speaker stream. This option is read by the `transcode` AP[5] where the current speaker is highlighted before/after transcoding. All the video streams are then merged into one video stream (one big picture is created) and thus have to be synchronized. Finally, the outgoing audio and video streams are also fully synchronized.




```
Project My_simple_MCU.first_attempt;
# project parameters (owner, notifications,
# overall resource requirements, ...)
{AP name="determine_speaker" ref=recognize_speaker1;
  # AP parameters
  inputs = audio_input(DIPRONN_INPUT(10002));
        # requested DiProNN input port is 10002
  outputs = output(sort_ARTP1.in);
  control_outputs =
              ctrl_out(my_VM1.video_distr.ctrl_in);
}
{VM  name="my_VM1" ref=my_VM1_image;
  # VM parameters
  {AP name="transcode" ref=transcode_video;
    inputs = in, stateshare_in;
   # stateshare_in is the input for
       # communication among parallel instances
    outputs = out(sort_ARTP2.in),
 stateshare_out(my_VM1.transcode.state_in);
    parallelizable;      # parallelizable AP
  } # ... other APs
}
{VM  name="my_VM2" ref=my_VM2_image;
  {AP name="sync_all" ref=syncer;
    inputs = audio_in, video_in;
    precision = 0.001;  # 1ms
    outputs = out(DIPRONN_OUTPUT(SEE_ARTP));
     # the real receiver inside ARTP packets
  } # ... other APs
} # ... other APs/VMs
```

Fig. 3. DiProNN session graph for simple MCU unit together with a fragment of relevant DiProNN program.

[4]The real method of determining current speaker is not important for this example. The possible methods of speaker recognition could be found in [8].

[5]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.

VI. Related Work

Thanks to lots of possible applications, the programmable networks principles became very popular and thus researched by lots of research teams. Thus, various architectures of active routers/nodes have been proposed—in this section we briefly describe only those ones mostly related to our work.

C&C Research Laboratories propose the CLARA—the prototype of a routing node in their JOURNEY network. The CLuster-based Active Router Architecture (CLARA, [9]) consists of a cluster of generic PCs connected by a fast System Area Network (the prototype implementation uses Myrinet network) providing customizing of media streams to the needs of their clients. The CLARA provides fixed functionality only and does not guarantee the processing of all packets sent—additional guarantees must be implemented end-to-end, according to the requirements of individual streams.

The LARA (Lancaster Active Router Architecture) [10] architecture encompasses both hardware and software active router design. The LARA++ (Lancaster's 2^{nd}-generation Active Router Architecture) [11], as the name indicates, evolved from the LARA. Against the LARA, which provided innovative hardware architecture, the LARA++ lays the main focus on the software design of the active router—its software architecture is designed to be largely independent of the underlying hardware and thus, it could run on a single-processor node as well as use a distributed architecture. However, both router architectures do not provide user-controlled arbitrary active programs uploading.

A Cluster-Based Active Router Architecture Supporting Audio and Video Stream Transcoding Service [12] is a project which presents a cluster-based active router implementation that provides audio and video transcoding service only. Similarly to the CLARA architecture, it is assumed that the media stream data can be divided into a sequence of media units that are ready for independent transcoding. The routing PC receives these media units from the sending user PC, and forwards them to the computing PCs for transcoding. Each computing PC independently processes the media units using the local computing resources and does not require any global stream state. Because of the limited resources available in the active router cluster, some packets in the media stream are sent out without being processed.

In comparison with the DiProNN, none of these architectures addresses promising virtualization technology with its benefits (but also with its problems), and tries to provide enhanced execution environment flexibility and stream applications programming comfortability.

VII. Conclusions and Future Work

In this paper, we have proposed a virtual machine oriented distributed programmable network node architecture named DiProNN. The main features of the DiProNN are that its users are able to upload their DiProNN session consisting of a set of their own active programs independent on each other and possibly running in their own virtual machines, and let their passing data being processed by the DiProNN. The communication among such active programs is provided using standard network services by machine hypervisor so that active programs are not forced to be aware of their neighbourhood. DiProNN cluster-based design also enables simultaneous parallel processing of active programs that are intended to run in parallel.

Concerning the current and future challenges, the proposed DiProNN architecture is being implemented based on the XEN virtual machine monitor. Further we also want to explore the mechanisms of QoS requirements assurance and scheduling mechanisms to be able to utilize DiProNN resources effectively. We want to explore all the three perspectives of DiProNN scheduling—scheduling active programs to virtual machines (when they do not have their own virtual machine specified), scheduling virtual machines to appropriate Processing units and scheduling active programs/virtual machines to suitable DiProNN nodes (when there are more DiProNN nodes on the path from a sender to a receiver, which are able to process given DiProNN session).

Acknowledgement

This project has been supported by research intent "Integrated Approach to Education of PhD Students in the Area of Parallel and Distributed Systems" (No. 102/05/H050).

References

[1] Jim E. Smith and Ravi Nair, *Virtual Machines: Versatile Platforms for Systems and Processes*, Elsevier Inc., 2005.

[2] Aravind Menon, Jose Renato Santos, Yoshio Turner, G. (John) Janakiraman, and Willy 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*, New York, NY, USA, 2005, pp. 13–23, ACM Press.

[3] Tomáš Rebok, "Active Router Communication Layer," Tech. Rep. 11/2004, CESNET, 2004.

[4] Eva Hladká and Zdeněk Salvet, "An Active Network Architecture: Distributed Computer or Transport Medium," in *Networking – ICN 2001: First International Conference Colmar, France, July 9-13, 2001, Proceedings, Part II*, P. Lorenz, Ed., Heidelberg, Jan. 2001, vol. 2094 of *Lecture Notes in Computer Science*, pp. 612–619, Springer-Verlag.

[5] Andrzej Cichocki, Marek Rusinkiewicz, and Darrell Woelk, *Workflow and Process Automation: Concepts and Technology*, Kluwer Academic Publishers, Norwell, MA, USA, 1998.

[6] William Thies, Michal Karczmarek, and Saman Amarasinghe, "Streamit: A language for streaming applications," in *International Conference on Compiler Construction*, Grenoble, France, Apr. 2002.

[7] Marc Willebeek-LeMair, Dilip D. Kandlur, and Zon-Yin Shae, "On multipoint control units for viodeoconferencing," in *LCN*, 1994, pp. 356–364.

[8] D. A. Reynolds, "An overview of automatic speaker recognition technology," in *Acoustics, Speech, and Signal Processing, 2002. Proceedings. (ICASSP '02). IEEE International Conference on*, 2002, vol. 4, pp. IV–4072–IV–4075 vol.4.

[9] Girish Welling, Maximilian Ott, and Saurabh Mathur, "A cluster-based active router architecture," *IEEE Micro*, vol. 21, no. 1, pp. 16–25, 2001.

[10] R. Cardoe, Joe Finney, Andrew C. Scott, and Doug Shepherd, "Lara: A prototype system for supporting high performance active networking," in *IWAN 1999*, 1999, pp. 117–131.

[11] S. Schmid, "LARA++ Design Specification," 2000, Lancaster University DMRG Internal Report, MPG-00-03, January 2000.

[12] Jiani Guo, Fang Chen, Laxmi Bhuyan, and Raj 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*, Washington, DC, USA, 2003, p. 44.2, IEEE Computer Society.