Distributed Synchronous Infrastructure for Multimedia Streams Transmission and Processing

Tomáš Rebok and Petr Holub

December 5, 2007

Abstract

During 2007, CESNET has built a synchronous infrastructure for multimedia streams transmission and processing, which is based on the DiProNN nodes. The report presents both the DiProNN nodes developed by CESNET and the infrastructure itself. We show a sample scenario it might be used for and test the infrastructure's behaviour and performance limitations.

Keywords: *distributed synchronous infrastructure, stream processing, multimedia content distribution, DiProNN, virtualisation, virtual machines.*

1 Introduction

Nowadays, multimedia content has become the major content transferred on computer networks. Since the distribution of non-realtime (so called *offline*) multimedia content does not have special demands on the computer network infrastructure used, the realtime multimedia streams requirements differ from the offline ones' requirements a lot. Realtime streams usually require stable and sometimes quite significant network bandwidth and as low end-to-end latency as possible.

However, mere content distribution itself may not be sufficient as processing of a realtime stream inside the network is often very desirable. For example, when streaming high-quality video stream using the HDV [1] format taking approximately 25 Mbps of network bandwidth, there might be users not having sufficient network connection capacity and thus some realtime transcoding to lower quality (which requires lower network bandwidth) must be done.

During 2007, CESNET has built a synchronous infrastructure for multimedia streams transmission and processing. The infrastructure is based on the *Distributed Programmable Network Nodes (DiProNN)*, that are developed by CESNET in cooperation with the Faculty of Informatics Masaryk University in Brno. In the infrastructure presented, the DiProNNs can be

used both for simple content distribution and/or arbitrary content processing.

In the first part, the report focuses on the architecture of the DiProNN nodes itself, while the next one describes the synchronous infrastructure built. Afterwards we describe a sample scenario the infrastructure might be used for, and finally we test and analyse the infrastructure's behaviour and performance limitations.

2 DiProNN: Distributed Programmable Network Node

2.1 Architecture

Single node DiProNN architecture, which has been proposed and discussed in [2], [3], [4], [5], [6], assumes the underlying infrastructure as shown in Figure 1. In this report, we give only very brief description and the reader who wants to find out more should look into the referenced papers. The DiProNN units form a computer cluster with each unit having two interconnections, at least on the conceptual level:

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

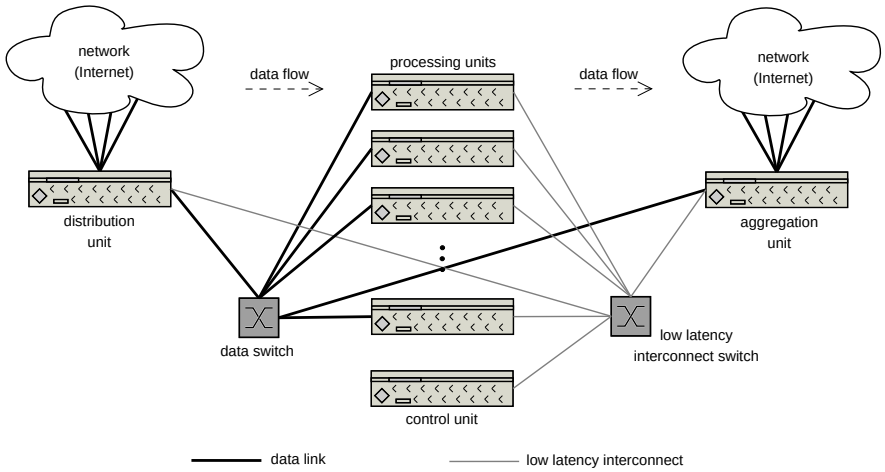

Figure 1: DiProNN single node architecture.

The low latency interconnection is desirable since current common network interfaces like Gigabit Ethernet provide large bandwidth, but the latency of the transmission is still in order of tens to hundreds of μs, which

is not suitable for fast synchronisation of DiProNN units. Thus, the use of specialised low-latency interconnects like Myrinet network providing as low latency as 10 μs (and even less, if you consider e.g., InfiniBand with 4 μs), 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, where they are forwarded to appropriate Processing unit(s) for processing. After the processing is completed, 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 2.1.1 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, synchronisation, processing control).

- **Control unit**—the Control unit is responsible for the whole DiProNN management and communication with its neighbourhood including communication with DiProNN users to negotiate new DiProNN sessions (details about DiProNN sessions establishment are given in Section 2.2) and, if requested, providing monitoring of their behaviour.

- **Aggregation unit**—the Aggregation unit collects the resulting traffic and sends it to the output network line(s). It serves as a simple forwarder forwarding incoming data from a DiProNN's private network segment to public Internet.

The DiProNN architecture presented represents the most general architecture possible. In fact, some DiProNN units might merge with the others (for example, the Distribution unit or one of the Processing units might perform as the Control unit). The most minimalistic DiProNN architecture consists of just single Processing unit which serves as all the DiProNN units simultaneously.

2.1.1 DiProNN Processing Unit Architecture

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 ensures strict separation of different virtual machines enhancing their security and also provides 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 [7]. 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.

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 (VMs) [8], 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.

2.1.2 Communication Protocol

For data transmission, the DiProNN users may use one of three transport protocols supported by DiProNN: the *User Datagram Protocol* (UDP) [9], the *Datagram Congestion Control Protocol* (DCCP) [10] and the transmission protocol called *Active Router Transmission Protocol* (ARTP, [11]) we

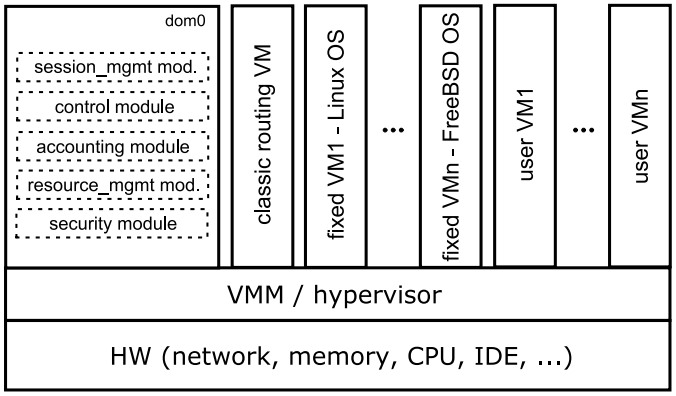

Figure 2: DiProNN Processing Unit Architecture.

originally designed and implemented for the generic active router architecture described in [12]. Depending on 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.

2.2 Programming Model

In this section we depict a programming model we propose for DiProNN programming. The DiProNN programming model is based on the workflow principles [13], and uses the idea of independent simple processing blocks, that composed into a processing graph constitute required complex processing. In DiProNN, the processing block is an active program and the communication among such active programs is thanks to the virtualisation mechanisms provided by machine hypervisor using common network services (details about DiProNN internal communication are provided in Section 2.2.1). The interconnected active programs then compose the *"DiProNN session"* described by its *"DiProNN session graph"*, which is a graphical representation of an *"DiProNN program"* (for example the ones used later, see Figure 5 and Figure 4). 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.

The DiProNN program defines active programs optionally with virtual machines they run in[1], 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

[1]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.

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 of 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.

2.2.1 Session Establishment and Data Flow

When a new DiProNN session request arrives to the node, the Distribution unit immediately forwards it to the 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 optimalisation) performed on another DiProNN node being on the path from the first DiProNN node to the receiver(s).

When the request can 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.

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). Using the information about registered couples together with the virtual machine and port number a packet is coming from, the control module properly sets the receiver of passing packets (using kernel `iptables` and its `DNAT` target). The packets 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. In this case, the Aggregation unit forwards these packets to the destination given inside ARTP datagram instead of the one given in DiProNN program.

3 Synchronous Infrastructure for Multimedia Streams Transmission and Processing

3.1 Overview

During the year 2007, CESNET has built an infrastructure for synchronous multimedia stream transmission and processing consisting of four DiProNN nodes located in Brno, Liberec, Pilsen, and Prague depicted in the Figure 3.

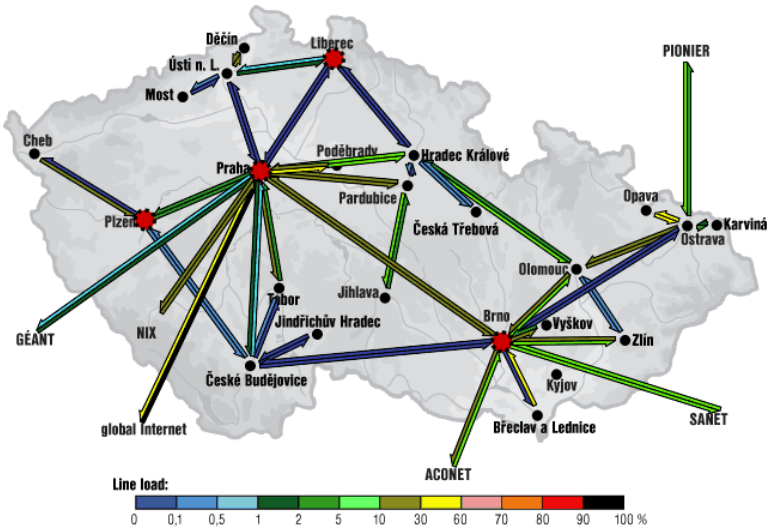

Figure 3: DiProNN nodes used for synchronous infrastructure (red dots).

All the DiProNN nodes are currently implemented using XEN Virtual Machine Monitor (VMM) [14]. The nodes are set using the DiProNN's minimal configuration possible—they consist of just single physical computer serving as the DiProNN Processing Unit, DiProNN Distribution Unit, DiProNN Aggregation Unit and DiProNN Control Unit simultaneously. All the four physical machines used for synchronous infrastructure have exactly identical hardware configuration given in Table 1.

The DiProNN nodes are interconnected with 1 GE network links using CESNET's public network infrastructure. In the future, we also plan to equip the nodes with Myrinet 10 GE network cards and test the infras-

	Configuration
Brand	Supermicro
Model	X7DBR-8
Processor	2× Core 2 Duo Intel Xeon 3.0 GHz
Front-side bus	533 MHz
Memory	4 GB DIMM DDR2
GE NIC	2× Intel PRO/1000 Network Adapter
Operating system	Linux Ubuntu 7.04 (Feisty Fawn) kernel 2.6.18-xen SMP

Table 1: Configuration of the DiProNN nodes used for synchronous infrastructure.

tructure behaviour when transmitting higher amounts of data, for example 1.5 Gbps High Definition (HDTV) [15] video streams, and/or jumbo packets much larger than current maximum of 1500 B.

3.2 Sample Scenario

To prove the infrastructure's proper functionality we decided to test it in an example realistic it might be used for.

Situation: Let's have a presentation taking place in Brno. The presentation has to be available for clients not being able to attend the presentation personally. For clients having high-bandwidth connection the presentation should be available in high quality HDV stream (generated in Brno using HDV cameras[2]), and for clients not having necessary capacity of their connections it should be transcoded in real-time and be available in lower quality. Both high quality and low quality streams have to be saved for later purposes, too.

However, since the video transcoding takes some time, at least the synchronisation of audio (not necessarily transcoded since it does not take such high network bandwidth as video does) and transcoded video streams must be done. Nevertheless, the synchronisation of the high quality audio and video streams is also highly desirable to prevent network fluctuations and possible desynchronisation of them.

We should point out, that even the original audio and video streams are also desynchronised because of the following reason: the HDV video stream outgoing from the HD camera is delayed by approximately 1 second, while the audio stream is captured by standalone audio grabber device with latency in order of tens of *ms*. Both the audio and non-transcoded video

[2]For our experiments we used Sony HVR-Z1R camera.

streams thus need to be synchronised, even if the network itself would not desynchronise them.

For the situation described we established[3] the DiProNN session described by its DiProNN session graph (Figure 4 and relevant DiProNN program (Figure 5). Both audio and video streams coming from their grabber devices were sent to the DiProNN node located in Brno (*V_in* input for video stream and *A_in* input for audio stream), where they were duplicated (*Dup_A* active program (AP) for audio duplication and *Dup_V* AP for video duplication). One *twinstream* (audio and video) was sent to Prague (high-quality stream), while the second one was forwarded to second VM running in Brno for transcoding (*Transcode* AP), and afterwards sent to Liberec. Both twinstreams (high-quality and low-quality) were then synchronised (*Sync_high* and *Sync_low* APs) and duplicated once again (*Dup_high* and *Dup_low* APs). One twinstream of each copy was sent to Pilsen and saved for later purposes (*Saver_high* and *Saver_low* APs), and the other one was sent to the reflector application [16], which served as the presentation content provider (having input/output ports named *in/out_high* and *in/out_low*) for all the connected clients (in fact, there were two reflectors—one for high-quality audio and video data and the other one for low-quality video and original audio data).

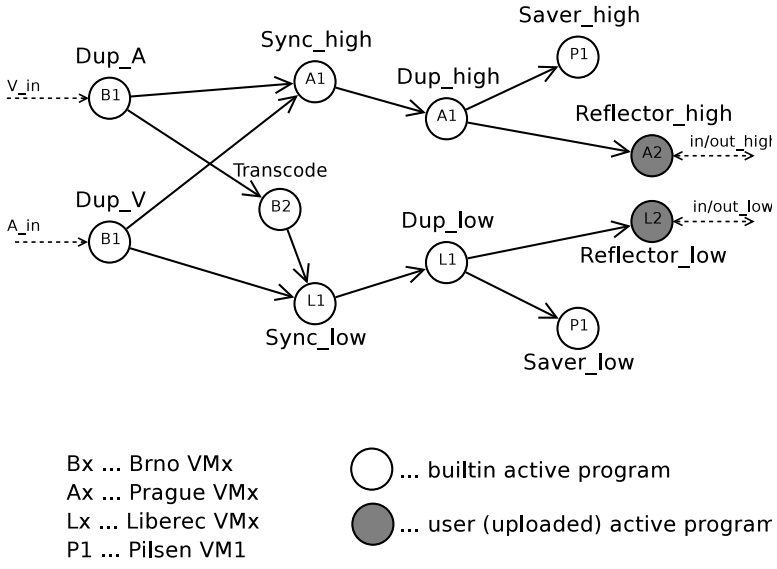

Figure 4: DiProNN session graph used in the example scenario.

[3]Since the DiProNN utilities serving for sessions' establishment (uploading, starting, and destroying) were not available at the moment, the whole DiProNN configuration was done manually. However, since the whole configuration can be simply derived from the DiProNN program, the utilities are quite simple and will be available during 2008.

The high-quality stream itself was transferred in HDV format and took about 25 Mbps of network bandwidth. For duplications, stream synchronisations and storage we used active programs built to DiProNN. As mentioned before, the content provider we used was our implementation of the reflector program [16], which was uploaded to DiProNN. The video stream generating, listening, and transcoding itself was done by the VLC media player[4]—the input HDV stream was transcoded into MP2V stream having variable bitrate (set to 256 kbps) and scaled down to 25 % of its original size. For audio capturing and listening the RAT application[5] set to Linear-16 codec in 16 kHz stereo mode was used. The audio stream thus took 512 kbps of the network bandwidth.

The overall client setup (located in Brno) is captured in the picture 6, and the detail of client's screen is captured in the picture 7. The client was connected to both high-quality and low-quality content providers, and thus the picture shows overall high-quality stream bandwidth (including all the IP and UDP headers it took about 30 Mbps), overall low-quality stream bandwidth (including headers it took about 820 kbps), and the latency taken by the transcoding itself (visible on the clocks streamed—about 1 second). The sizes of files containing saved 20 minutes streams[6] were 3,2 GB for high-quality video stream, 103 MB for transcoded video stream and 79 MB for each audio stream (saved twice—once as a high-quality twinstream and once as a low-quality twinstream).

4 Infrastructure behaviour tests

To test the behaviour and performance limitations of the synchronous infrastructure presented we kept the setup used for previous experiments[7] and generated streams of UDP data of four different packet sizes (100 B, 500 B, 1000 B, and 1450 B). All the streams were sequentially sent from separate PC having configuration given in Table 2 through the HP Procurve 6108 switch to input video port of the DiProNN node located in Brno, while analysing applications (running on the same machine as the stream generator) were simultaneously connected to both content providers (Prague and Liberec). Furthermore, two others analysing applications connected to both content providers were also running in Pilsen (on the DiProNN node used for data storage in previous experiment).

[4] `http://www.videolan.org/vlc`

[5] `http://mediatools.cs.ucl.ac.uk/nets/mmedia`

[6] The streams were saved in simple packet form—the whole UDP packet content and its timestamp were saved in separate files for both audio and video streams. We have also created the player that is able to read such file content and sent it to the network like it would be sent in real-time.

[7] In fact, there were a few changes done—the transcoding and synchronising modules were replaced by simple packet forwarders.

	Configuration
Processor	Intel Core 2 Duo CPU, 2.66 GHz
Memory	2 GB DIMM DDR2
GE NIC	Intel PRO/1000 Network Adapter
Operating system	Linux Ubuntu 6.10 (Edgy Eft) kernel 2.6.20 SMP

Table 2: Configuration of the tester machine generating the UDP test data.

All the DiProNN nodes were running two VMs (except the Pilsen node which was running single VM) with exactly identical configuration given in Table 3.

	Virtual Machine		
	dom0	dom1	dom2
#CPU	2	1	1
Memory	1.5 GB	1 GB	1 GB
OS	Ubuntu 7.04	Ubuntu 7.04	Ubuntu 7.04
Kernel	2.6.18-xen SMP	2.6.18-xen SMP	2.6.18-xen SMP
Scheduler	credit		

Table 3: The configuration of DiProNN nodes used.

The bandwidth of generated UDP data stream of given packet size was gradually incremented, and the analysers in Brno determined the packet loss and delay, while the analysers in Pilsen determined the packet loss only. The packet delay in Pilsen could not be easily determined since the sending machine and Pilsen DiProNN node had their clocks desynchronised. The graphs in Figures 8, 9, 10 and 11 show the results achieved.

4.1 Results discussion

Before we discuss the results, it is necessary to point out that the theoretical maximum achievable during our experiments was 500 Mbps since the streams were duplicated on the DiProNN nodes and were sent through the network lines having maximum capacity of 1 Gbps twice. Thus at least the Brno output lines, Brno input lines (analysers) and Pilsen input lines would be saturated.

However, the maximal bandwidth of 500 Mbps is highly theoretical, since the real network lines usually do not provide their theoretical capacity. Thus, we have measured the real maximal bandwidths achievable on synchronous infrastructure built. The maximal throughputs were measured between each two nodes (their *dom0*s and their *domU*s) directly communicating

in DiProNN session described above. The data streams were generated by the `iperf` tool, and the results achieved are summarised in Tables 4 and 5.

Packet size	*Maximal throughput [Mbps]*			
	Br-Pra	Br-Lib	Pra-Pil	Lib-Pil
100 B	159	124	169	134
500 B	597	582	603	599
1000 B	923	889	929	907
1450 B	937	918	944	924

Table 4: Maximal throughputs achieved (between *dom0*s).

Packet size	*Maximal throughput [Mbps]*			
	Br-Pra	Br-Lib	Pra-Pil	Lib-Pil
100 B	146	108	162	121
500 B	577	578	592	586
1000 B	901	866	916	891
1450 B	923	899	925	903

Table 5: Maximal throughputs achieved (between *domU*s).

The comparison of Tables 4 and 5 shows, that the virtualisation itself has almost no impact on the infrastructure's network performance. The next table (Table 6) shows, that the maximal bandwidths achievable during tested DiProNN session, that are restricted by the network itself (not by the XEN and its internal communication).

It is obvious, that the network itself is not the bottleneck of the DiProNN session tested during our example scenario, and thus we have to keep looking for the real bottleneck to make the achieved results clear.

Since the internal XEN communication among *dom0* and *domU*s can bring other performance overheads, we have tested the effect of network stream multiplication on the most burdened DiProNN node—Brno node. It is obvious, that during the DiProNN session tested the particular VMs running on the Brno node had to cope with input/output bandwidths given in Table 7.

The Table 7 shows that when having the input node bandwidth x (for example 100 Mbps), the *dom0* had to cope with the input network bandwidth of size $4x$ (400 Mbps) while each *dom_U* had to cope with the input network bandwidth of size x (100 Mbps). The output *dom0* bandwidth was also $4x$ ($2x$ to *dom_U*s and $2x$ out of the node), while the output bandwidth of *domU_1* was $2x$ and the output bandwidth of *domU_2* was x.

Thus we have tried to saturate the Brno *dom0* with up to four network

Packet size	*Throughput [Mbps]*			
	Prague	Liberec	Pilsen from Pra	Pilsen from Lib
100 B	74.5	62	74.5	62
500 B	298.5	291	298.5	291
1000 B	461.5	444.5	461.5	444.5
1450 B	468.5	459	468.5	459

Table 6: The maximal throughput that was achieved during performance tests (restricted by the network itself). The maximal achievable bandwidths between *dom0*s (Table 4 are divided by 2 and the minimal value on the path from the data sender to the data receiver connected to given content provider is presented. The reason why we use the maximal bandwidth between *dom0*s and not between *domU*s is, that DiProNN always sends all the data to *dom0*s where they are forwarded using `iptables` rules to appropriate active programs for further processing.

	Bandwidth
input node BW	x
domU_1 from dom0	x
domU_2 from dom0	x
dom0 from domU_1	$2x$
dom0 from domU_2	x
output node BW	$2x$

Table 7: The study of input/output bandwidths of the Brno DiProNN node.

streams generated from the *dom_U*s on the same host. The *dom0* had to return the streams back to the sender where the maximal throughput was analysed[8]. We have tested same packet sizes as we did in previous tests—100 B, 500 B, 1000 B and 1450 B. The results achieved are summarised in the Table 8.

Results conclusions

Concerning the infrastructure tests, the approximate limitations of the whole infrastructure not being set to any DiProNN session were presented in the Table 6. The results show that the virtualisation used has almost no impact on the infrastructure network performance and thus the DiProNN nodes are able to communicate with each other as if they were running native Linux OS.

[8]Note, that during these tests the maximal bandwidth available was not limited by the 1 Gbps network interconnection—all the packets were sent among domains using the XEN internal interconnection.

Packet size	*Average throughput [Mbps]*			
	1 stream	2 streams	3 streams	4 streams
100 B	86.5	64.5	44.2	34.1
500 B	398.4	291.1	218.7	163.0
1000 B	778.2	407.9	324.0	275.4
1450 B	828.8	460.1	411.1	365.1

Table 8: The maximal throughputs achieved on the Brno DiProNN node (internal XEN communication).

To find out how the infrastructure behaves while set to typical DiProNN session we tried to analyse the theoretical results achievable, and tried to find out the reason why the DiProNN session tests behave in the way we measured. We found out, that for the scenario we tested, the Brno node has to cope with the highest amount of data, and thus was the bottleneck of the whole DiProNN session. The comparison of the Table 8 with the graphs in the Figures 8, 9, 10 and 11 shows, that during the DiProNN session tests we have reached the maximal capacity of the Brno DiProNN node. However, the infrastructure performance depends a lot on the scenario it is used for—for simple scenarios its performance might be limited solely by the network performance, while for more complicated ones the performance might decrease a lot.

5 Conclusions

In this report we have presented the synchronous infrastructure for multimedia streams transmission and processing built during the year 2007 by CESNET. The architecture of the infrastructure nodes, based on the DiProNN nodes being developed by CESNET in cooperation with the Faculty of Informatics Masaryk University in Brno, was also presented. Besides that the report shows the typical scenario the infrastructure can be used for. Furthermore, we have presented several tests demonstrating the behaviour and performance limitations of the infrastructure set to the scenario it will be mainly used for.

Regarding our future work, the infrastructure built is supposed to be used for scenarios similar to the tested one. We will analyse the most used scenarios and study their bottlenecks, and if necessary, we will try to optimise them. Another interesting topic for our future work is to implement Quality of Service (QoS) assurances in the DiProNN nodes and try to keep the network parameters within desired limits. Furthermore, we will try to equip at least some of the DiProNN nodes by Myrinet 10 Gbps network cards and study the infrastructure behaviour when it is used for higher network

bandwidths and/or transmitting bigger packets (for example jumbo packets having 8500 B instead of standard 1500 B).

References

[1] *HDV Video Format.* `http://www.hdv-info.org`

[2] Tomáš Rebok. *DiProNN: Distributed Programmable Network Node Architecture.* In Proceedings of The Fourth International Conference on Networking and Services (ICNS'08), Gosier, Guadeloupe, accepted paper.

[3] Tomáš Rebok. *DiProNN: Distributed Programmable Network Node Architecture.* In Proceedings of the Cracow Grid Workshop (CGW'07), Cracow, Poland, 2007.

[4] Tomáš Rebok. *VM-based Distributed Active Router Design.* In Proceedings of the European Computing Conference (ECC'07), Athens, Greece, 2007.

[5] Tomáš Rebok. *DiProNN Programming Model.* In Proceedings of the MEMICS'07 conference, Znojmo, Czech Republic, 2007.

[6] Tomáš Rebok. *DiProNN: VM-based Distributed Programmable Network Node Architecture.* TERENA'07 Networking Conference poster, Copenhagen, Denmark, 2007.

[7] Aravind Menon and Jose Renato Santos and Yoshio Turner and G. (John) Janakiraman and Willy Zwaenepoel. *Diagnosing performance overheads in the XEN virtual machine environment.* VEE '05: Proceedings of the 1st ACM/USENIX international conference on Virtual execution environments, Chicago, USA, 2005.

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

[9] *User Datagram Protocol (UDP).* RFC 768, `ftp://ftp.isi.edu/in-notes/rfc768.txt`.

[10] *Datagram Congestion Control Protocol (DCCP).* RFC 4340, `http://www.rfc-editor.org/rfc/rfc4340.txt`.

[11] Tomáš Rebok. *Active Router Communication Layer.* Technical report, 29 pages, CESNET, Prague, 2004.

[12] Eva Hladká and Zdeněk Salvet. *An Active Network Architecture: Distributed Computer or Transport Medium.* In Proceedings of the ICN

2001: First International Conference Colmar, pages 612-619, France, July, 2001.

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

[14] Boris Dragovic and Keir Fraser and Steven Hand and Tim Harris and Alex Ho and Ian Pratt and Andrew Warfield and Paul Barham and Rolf Neugebauer. *Xen and the Art of Virtualization.* In Proceedings of the ACM Symposium on Operating Systems Principles, Bolton Landing, NY, USA, 2003.

[15] John Ive. *Image Formats for HDTV.* Sony Europe – PSE, technical report.

[16] Eva Hladká and Petr Holub and Jiří Denemark. *User Empowered Virtual Multicast for Multimedia Communication.* In ICN'2004 Conference Proceedings, 2004.

```
Project synchro_infrastructure.first_tests;
owner = "Tom Rebok"
notifications = none
use_RTP;  # means, that DiProNN will suppose two ports for each
            interconnection (port and port+1)
{ AP name="Dup_V" ref=localservice.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=localservice.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=localservice.transcoder;
    inputs = in;
    outputs = out(Sync_low.in1);
    output_format = "mp4v";
    bitrate = "variable(256)";
    scale = "0.25"
}
{ AP name="Sync_high" ref=localservice.syncer;
    inputs = in1, in2;
    outputs = out(Dup_high.in);
    precision = 0.001;  # 1ms
}
{ AP name="Sync_low" ref=localservice.syncer;
    inputs = in1, in2;
    outputs = out(Dup_low.in);
    precision = 0.001;  # 1ms
}
...  # Dup_high and Dup_low duplicators defined similarly as above
{ AP name="Saver_high" ref=localservice.saver;
    inputs = in;
    output_file = "stream_high.dump";
}
{ AP name="Saver_low" ref=localservice.saver;
    inputs = in;
    output_file = "stream_low.dump";
}

{ VM  name="my_VM1" ref=my_VM1_image;
    { AP name="Reflector_high" ref=reflector;
        inputs = in, participant_registration(DIPRONN_INPUT(12345));
        outputs = out(DIPRONN_OUTPUT),
    }
    { AP name="Reflector_low" ref=reflector;
        inputs = in, participant_registration(DIPRONN_INPUT(12354));
        outputs = out(DIPRONN_OUTPUT),
    }
}
```

Figure 5: DiProNN program used in the example scenario.

Figure 6: Client side setup.

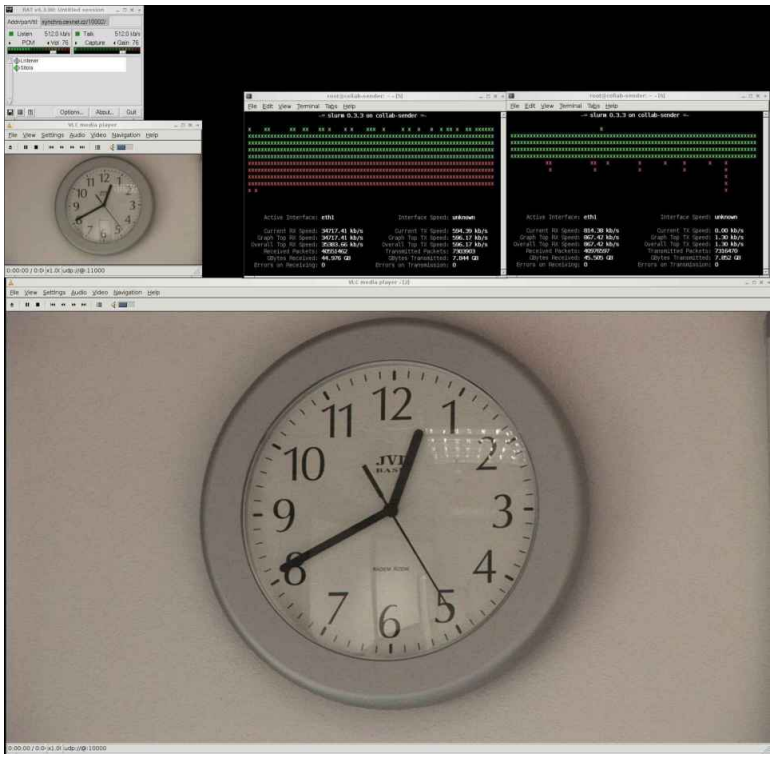

Figure 7: Client side screenshot. The clocks' difference shows the latency given by the transcoding itself (approximately 1 second).

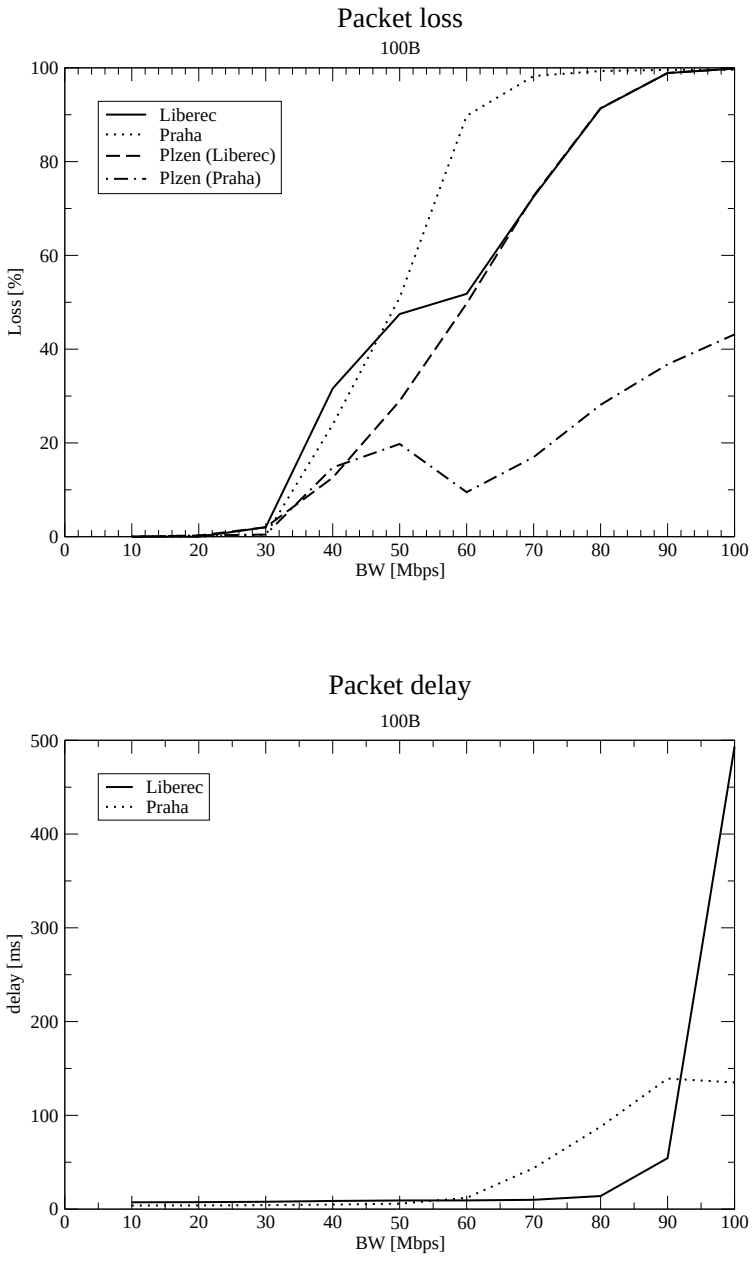

Figure 8: Achieved results for packet size 100 B.

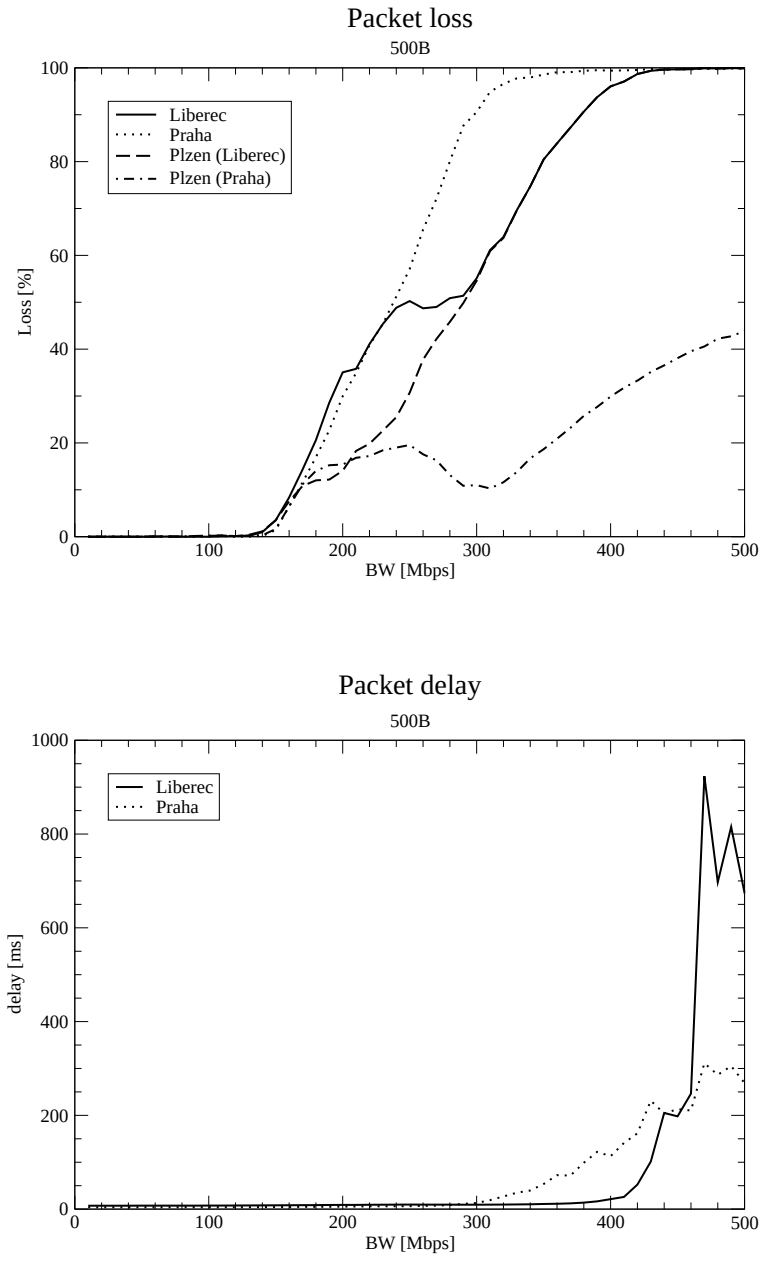

Figure 9: Achieved results for packet size 500 B.

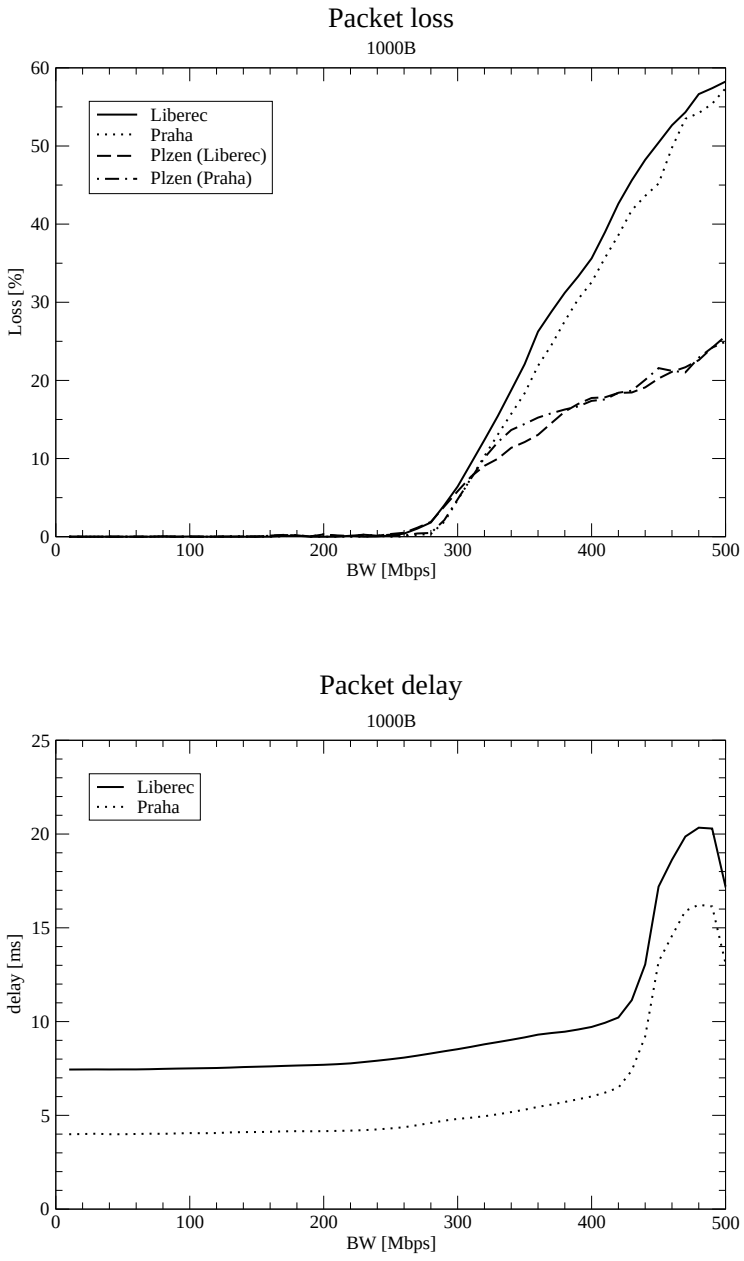

Figure 10: Achieved results for packet size 1000 B.

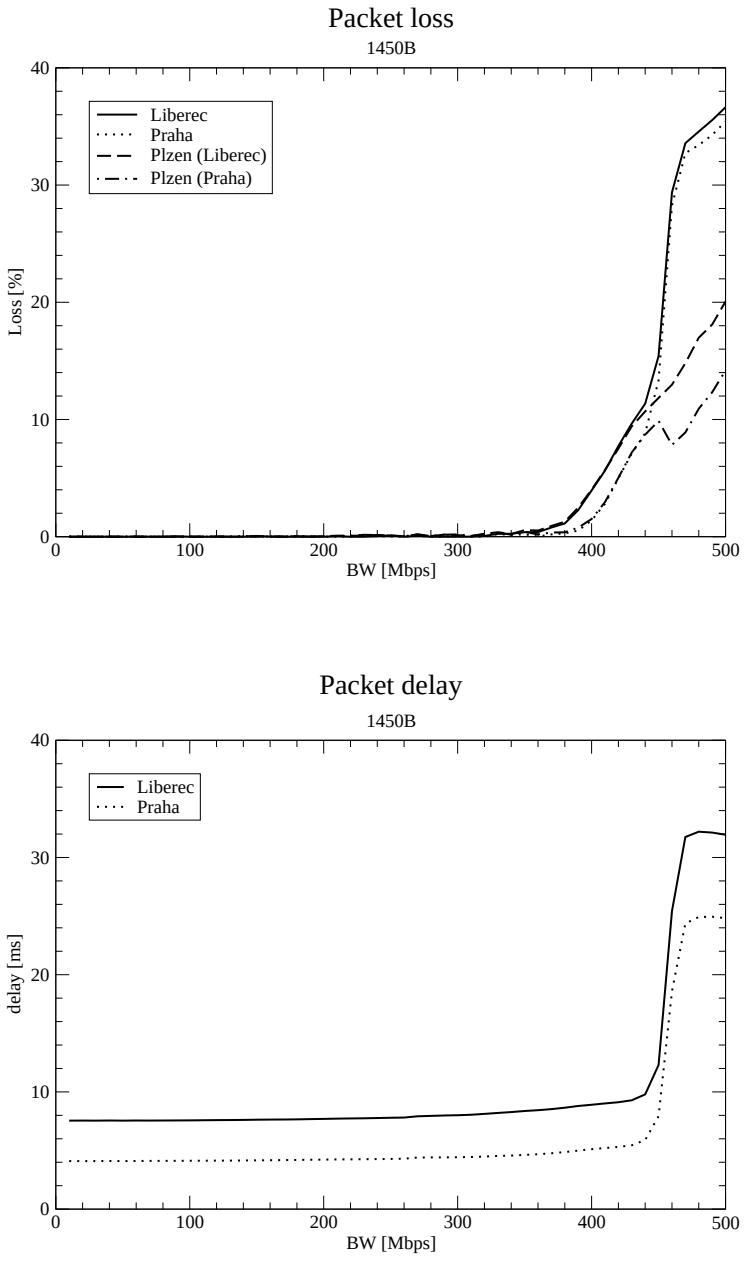

Figure 11: Achieved results for packet size 1450 B.