DiProNN: Distributed Programmable Network Node Architecture

Tomáš Rebok[1]

Faculty of Informatics
Masaryk University
Brno, Czech Republic
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.

1 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 processing of passing user data in a network, which is highly suitable especially for video 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).

The usage of virtual machines principles [1] can improve the flexibility of programmable nodes' execution environment since they are able to run completely different execution environments simultaneously. Moreover, they can also bring other benefits (strong isolation, resource management, programming flexibility, etc.), and thus make the usage of programmable routers easier.

In this paper we present 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 based on the workflow principles that takes advantages of DiProNN virtualization and makes programming of complex streaming applications more comfortable. Thanks to DiProNN's distributed architecture and possibilities of parallel processing, the DiProNN can also improve the robustness and scalability of such an active system with respect to number of active programs simultaneously running on the node and with respect to the bandwidth of each passing stream processed. As a possible application we show an implementation of simple MCU (Multipoint Control Unit) used for large videoconferences that profits from DiProNN properties.

2 DiProNN: Distributed Programmable Network Node

2.1 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 and synchronization inside the DiProNN, and at least one[1] *data connection* used for receiving and sending data.

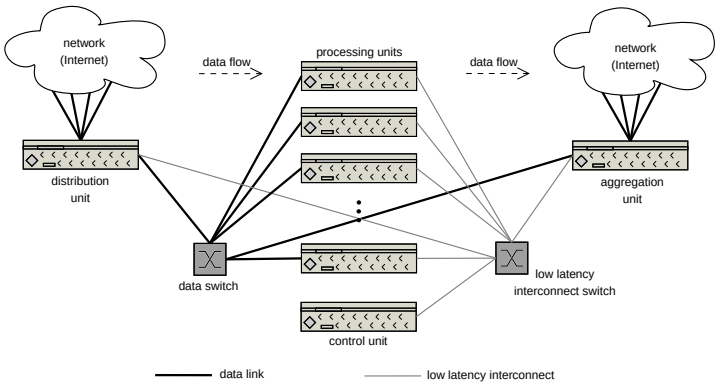


Fig. 1: Proposed DiProNN architecture.

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 μs

[1] The ingress data connection could be the same as the egress one.

(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 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 2.2 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 3) and, if requested, providing feedback about their behavior.
- **Aggregation unit**—the Aggregation unit aggregates the resulting traffic to the output network line(s).

2.2 DiProNN Processing Units

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.

2.2.1 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.

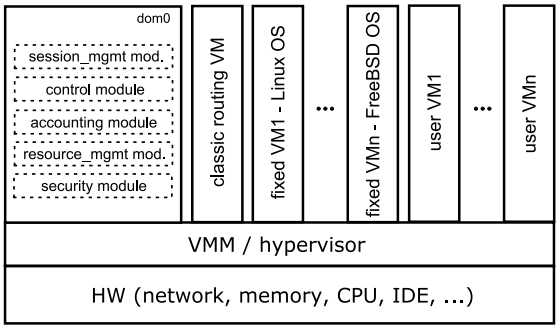


Fig. 2: DiProNN Processing Unit Architecture

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.

2.3 DiProNN 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])—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. 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 or not.

3 DiProNN Programming Model

The programming model we propose for DiProNN programming is based on the workflow principles [4] and was inspired by the idea of the StreamIt [5], 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 standard network services (details about DiProNN internal communication are provided in Section 4). 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 5.

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 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.

4 Session Establishment and Data Flow

When a new DiProNN session request arrives to the node, the Control unit decides, whether it could be satisfied or not (depending on actual DiProNN usage). 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

[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.

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. In this case, the Aggregation unit forwards these packets to the destination given inside ARTP datagram instead of the one given in DiProNN program.

5 Example: Simple MCU Unit in DiProNN

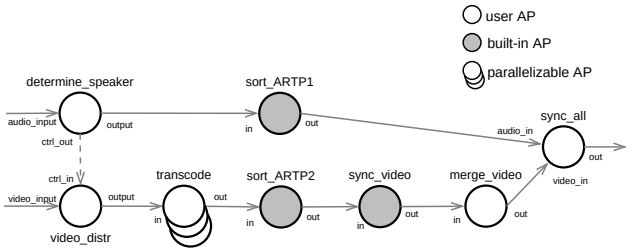
In this section we sketch a possible implementation of simple MCU unit (Multipoint Control Unit, [6]) used for videoconferencing. The MCU unit we want to implement should have following functionality: the unit should be able to accept up to twelve input ARTP streams of audio and video data (e.g., 6 audio 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[3] 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[4] 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

[3]The real method of determining current speaker is not important for this example. The possible methods of speaker recognition could be found in [7].

[4]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.

audio and video streams are also fully synchronized.




```
Project My_simple_MCU.first_attempt;
# project parameters (owner, notifications, 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 ... 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.

6 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 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, 2005, USA.
3. Tomáš Rebok. *Active Router Communication Layer.* Technical report, 29 pages, CESNET, Prague, 2004.
4. Andrzej Cichocki and Marek Rusinkiewicz and Darrell Woelk. *Workflow and Process Automation: Concepts and Technology.* Kluwer Academic Publishers, Norwell, MA, USA, 1998.
5. William Thies and Michal Karczmarek and Saman P. Amarasinghe. *StreamIt: A Language for Streaming Applications.* In Proceedings of the 11th International Conference on Compiler Construction, pages 179-196, 2002.
6. Willebeek-LeMair, M.H. and Kandlur, D.D. and Shae, Z.-Y. *On multipoint control units for videoconferencing.* Local Computer Networks, 1994. Proceedings, 19th Conference on. Minneapolis, MN, USA, pages 356-364, 1994.
7. Douglas A. Reynolds and Larry P. Heck. *Automatic Speaker Recognition: Recent Progress, Current Applications, and Future Trends.* Presentation at the AAAS 2000 Meeting Humans, Computers and Speech Symposium, 2000.