DiProNN Programming Model

Tomáš Rebok

Faculty of Informatics
Masaryk University
Botanická 68a, 602 00 Brno
xrebok@fi.muni.cz

Abstract. The programmable network approach allows processing of passing user data in a network, which is highly suitable especially for video streams processing. However, the 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 propose a programming model for our DiProNN node—the programmable network node that is able to accept and run user-supplied programs and/or virtual machines and process them over passing user data. The proposed modular programming model is based on the workflow principles and takes advantages of DiProNN virtualization, thanks to which the programming of complex streaming applications is much easier since they might be separated into several single-purpose simple programs. As an example application we show an implementation of simple MCU (Multipoint Control Unit) 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 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.

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 capable of running it, and thus the original active program has to be revised.

The usage of virtual machines principles [1] can increase a lot 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.

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 data. All the DiProNN programs are described using novel modular programming model described in this paper. The model is based on the workflow principles and takes advantages of DiProNN virtualization, thanks to which the programming of complex streaming applications in DiProNN is highly comfortable. As an example 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

DiProNN architecture we propose assumes the infrastructure as shown in Figure 1. The computing nodes form a computer cluster interconnected with each node 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.

Further details about the whole DiProNN architecture as well as the architecture of all the DiProNN components can be found in [2], [3] or [4].

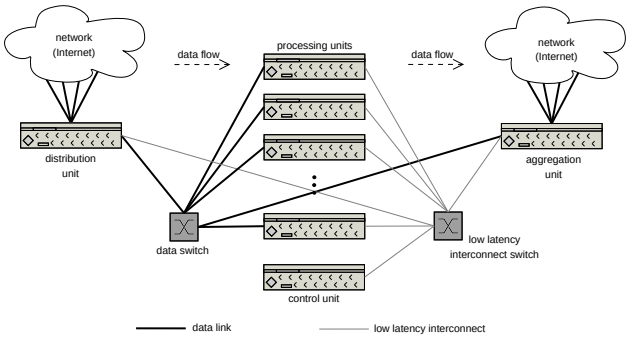


Fig. 1. Proposed DiProNN architecture.

3 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 is similar to the idea of the StreamIt [6], which is a language and compiler specifically designed for modern stream programming.

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

For the DiProNN programming model we adopted the idea of independent simple processing blocks (*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 3.2). 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). To achieve desired level of abstraction all the active programs as well as the input/output data/communication interfaces are reffered by their hierarchical names as shown in the MCU example in Section 4.

3.1 DiProNN Program

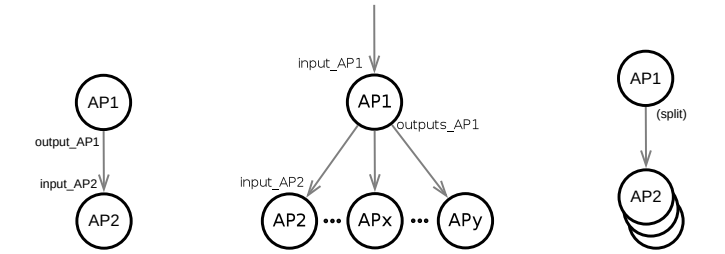
The DiProNN program defines active programs optionally with virtual machines they run in[2], which are neccessary for DiProNN session processing, and defines both data and control communication among them. Besides that, the DiProNN program may also define other parameters of active programs or whole DiProNN session and/or resource requirements they have for proper functionality.

Data Interconnection Constructs As mentioned before, the DiProNN program defines active programs required for DiProNN session processing, and data and control communication among them. Besides that, the DiProNN program consists of following constructs describing communication channels and active programs' processing:

- **Serialize** – as its name indicates, this construct is used to serialize processing of two independent active programs. It means, that the output(s) of one active program is/are connected to the input of another one(s). An example is given in the Figure 2(a).
- **Split** – the Split construct is used to specify independent parallel streams, which are further processed separately, and specifies how items from the input are distributed over the parallel components. There can be both built-in general splitters (e.g., duplication or RoundRobin function) and user-defined (and thus uploaded) splitters. An example is given in the Figure 2(b).
- **Parallelize** – if the special attribute of an active program (*parallelizable*) is set, the DiProNN performs its processing in parallel (see Figure 2(c)). The number of parallel instances running can be either fixed (set in DiProNN program and negotiated during DiProNN session establishment) or variable (controlled by the Control unit depending on actual active program usage and resources available).

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

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 Split construct (built-in or user-loaded) defined.



(a) Serialize (b) Split (c) Parallelize

Fig. 2. DiProNN program constructs.

Internal Control Communication Sometimes two or more active programs processing given data stream have to communicate with each other (to indicate some new event occured 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.

Nevertheless, the real control messages are not sent via data links as the data are, but via internal low-latency interconnection described in Section 2 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.

3.2 DiProNN Session Establishement and Data Flow

When a new DiProNN session request arrives to the node, the Distribution unit immediatelly forwards it to the Control unit. If 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)[3], 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).

[3] Different parts of a whole processing might be performed on different DiProNN nodes for resource or network optimization.

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 and the Control unit of each DiProNN node decide, 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 appropriatery. Moreover, all the requested resources are reserved, if any.

Once the session is established, the data flow through each DiProNN node could be briefly described in the following way: when a packet arrives to the Distribution unit, it is forwarded to the first active program processing given DiProNN session (to an interface of appropriate virtual machine and port the active program listens on), where in case of ARTP protocol[4] it is reassembled into original ARTP datagram and processed (when using UDP protocol it is processed directly). After the processing, the ARTP datagram is fragmented again into ARTP packets, which are forwarded to next active program(s) for further processing (in case of UDP protocol packets are forwarded directly). Finally, all the packets are forwarded to the Aggregation unit, where they are sent away to their receiver or to next processing node.

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 the couple (symbolic name, real port) is using simple text protocol registered. The control module using the information about registered couples together with the DiProNN program a packet is coming from properly sets the IP receiver of passing packets, which are then automatically forwarded to proper active programs for processing.

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 (e.g., using proper option inside each ARTP datagram). In this case, the Aggregation unit forwards these packets to the destination given inside ARTP datagram instead of the one(s) given in DiProNN program.

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 (using simple text protocol) at the control module of the Processing unit they run in.

[4] The *Active Router Transmission Protocol* (see [7]).

4 Example: Simple MCU Unit in DiProNN

In this section we briefly describe a possible implementation of simple MCU unit (Multipoint Control Unit, [8]) used for videoconferencing. The MCU unit we want to sketch should be able to accept up to twelve input ARTP streams of audio and video data (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[5] using the `determine_speaker` active program. The identification of current speaker is then using low-latency control interconnection sent to the `video_distr` AP (the user-defined Split construct as described in Section 3.1), where relevant ARTP option indicating current speaker is added to the speaker stream. This option is read by the `transcode` AP[6] 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 must be synchronized. Finally, the outgoing audio and video streams are also fully synchronized.

5 Related Work

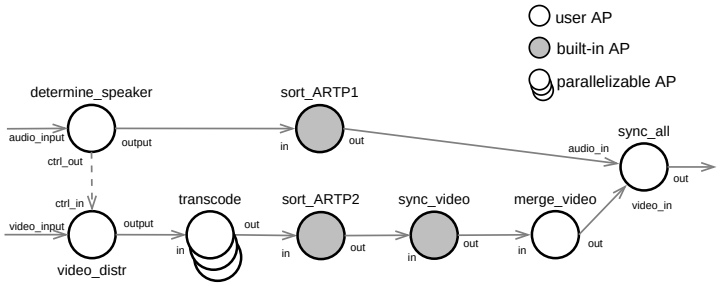
Thanks to lots of possible applications, the programmable networks principles are very popular and thus researched by lots of research teams. Thus, various architectures of programmable routers/nodes have been proposed—in this section we briefly remind only those ones mostly related to our work.

C&C Research Laboratories propose the CLARA (CLuster-based Active Router Architecture, [10]) providing customizing of media streams to the needs of their clients. The architecture of another programmable network node, LARA (Lancaster Active Router Architecture, [11]) in comparison with CLARA encompasses both hardware and software active router design. The LARA++ (Lancaster's 2nd-generation Active Router Architecture, [12]), 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 its architecture.

However, in comparison with the DiProNN, none of these distributed programmable architectures addresses promising virtualization technology and its benefits, and tries to provide enhanced flexibility (both programming flexibility and execution environment flexibility).

[5] The real method of determining current speaker is not important for this example. The possible methods of speaker recognition could be found in [9].

[6] Note, that the `transcode` AP might be processed in parallel as indicated in the DiProNN program.




```
Project My_simple_MCU.first_attempt;
# project parameters (owner, notifications,
# overall resource requirements, ...)
{AP name="determine_speaker" ref=recognize_speaker1;
  # AP parameters
  inputs = DIPRONN_INPUT(10002);
        # requested DiProNN input port is 10002
  output = sort_ARTP1.in;
  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;
    output = sort_ARTP2.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
    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 shown the programming model we use for our Distributed Programmable Network Node (DiProNN). Thanks to the main features of the DiProNN—the VM-based design and possibilities of uploading both the virtual machines and active programs itself—the modular programming model we propose makes DiProNN programming more comfortable since the complex func-

tionality required to be done on the node can be separated into several single-purpose active programs with the data and control flow among them defined.

Concerning the future challenges, the proposed DiProNN architecture is being implemented based on the Xen virtual machine monitor [13]. Further we want to explore the mechanisms of QoS requirements assurance and scheduling mechanisms to be able to utilize DiProNN resources effectively.

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. Smith, J.E., Nair, R.: Virtual Machines: Versatile Platforms for Systems and Processes. Elsevier Inc. (2005)
2. Rebok, T.: Vm-based distributed active router design. In: MEMICS'06 Conference Proceedings. (2006) 190–197
3. Rebok, T.: DiProNN: VM-based Distributed Programmable Network Node Architecture (2007) TERENA'07 Networking Conference poster.
4. Rebok, T.: Vm-based distributed active router design. In: European Computing Conference (ECC'07) Conference Proceedings. (2007) Accepted paper (not published yet).
5. Cichocki, A., Rusinkiewicz, M., Woelk, D.: Workflow and Process Automation: Concepts and Technology. Kluwer Academic Publishers, Norwell, MA, USA (1998)
6. Thies, W., Karczmarek, M., Amarasinghe, S.: Streamit: A language for streaming applications. In: International Conference on Compiler Construction, Grenoble, France (2002)
7. Rebok, T.: Active Router Communication Layer. Technical Report 11/2004, CESNET (2004)
8. Willebeek-LeMair, M., Kandlur, D.D., Shae, Z.Y.: On multipoint control units for viodeoconferencing. In: LCN. (1994) 356–364
9. Reynolds, D.A.: An overview of automatic speaker recognition technology. In: Acoustics, Speech, and Signal Processing, 2002. Proceedings. (ICASSP '02). IEEE International Conference on. Volume 4. (2002) IV–4072–IV–4075 vol.4
10. Welling, G., Ott, M., Mathur, S.: A cluster-based active router architecture. IEEE Micro **21**(1) (2001) 16–25
11. Cardoe, R., Finney, J., Scott, A.C., Shepherd, D.: Lara: A prototype system for supporting high performance active networking. In: IWAN 1999. (1999) 117–131
12. Schmid, S.: LARA++ Design Specification (2000) Lancaster University DMRG Internal Report, MPG-00-03, January 2000.
13. Dragovic, B., Fraser, K., Hand, S., Harris, T., Ho, A., Pratt, I., Warfield, A., m, P.B., Neugebauer, R.: Xen and the Art of Virtualization. In: Proceedings of the ACM Symposium on Operating Systems Principles, Bolton Landing, NY, USA (2003)