DiProNN: VM-based Distributed Programmable Network Node Architecture

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 present the architecture of our DiProNN node—the VM-based Distributed Programmable Network Node, that is able to accept and run user-supplied programs and/or virtual machines and process them (in parallel if requested) over passing user data. The node is primarily meant to perform stream processing—to enhance DiProNN flexibility and make programming of streaming applications for DiProNN node easier, we also propose suitable modular programming model which takes advantages of DiProNN virtualization and makes its programming more comfortable.

1 Introduction

Contemporary computer networks behave as a passive transport medium which delivers (or in case of the best-effort service tries to deliver) data from the sender to the receiver. The whole transmission is done without any modification of the passing user data by the internal network elements[1]. However, especially for small and middle specialized groups (e.g., up to hundreds of people) using computer networks for specific purposes, the ability to perform a processing inside a network is sometimes highly desired.

The principle called "Active Networks" or "Programmable Networks" is an attempt how to build such intelligent and flexible network using current "dumb and fast" networks serving as a communication underlay. In such a network, users and applications have the possibility of running their own programs inside the network using inner nodes (called *active nodes/routers*, or *programmable nodes/routers*—all with rather identical meaning) as processing elements.

However, the programming of complex (active) programs used for data processing, that are afterwards uploaded on programmable nodes, may be fairly

[1] Excluding firewalls, proxies, and similar elements, where an intervention is usually limited (they do not process packets' data).

difficult. And usually, if such programs exist, they are designed for different operating systems and/or use specialized libraries than the programmable node provides as an execution environment. Furthermore, since the speeds of network links still increase and, subsequently, applications' demands for higher network bandwidths increase as well, a single programmable node is infeasible to process passing user data in real-time, since such processing may be fairly complex.

The main goal of our work is to propose a distributed programmable network node architecture with loadable functionality that uses commodity PC clusters interconnected via the low-latency interconnection (so called tightly coupled clusters), which is also able to perform distributed processing. Since the node is primarily meant to perform stream processing, and to make programming of stream-processing applications for our node easier, we also propose suitable modular programming model which takes advantages of node virtualization and makes its programming more comfortable.

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[2] *data connection* used for receiving and sending data.

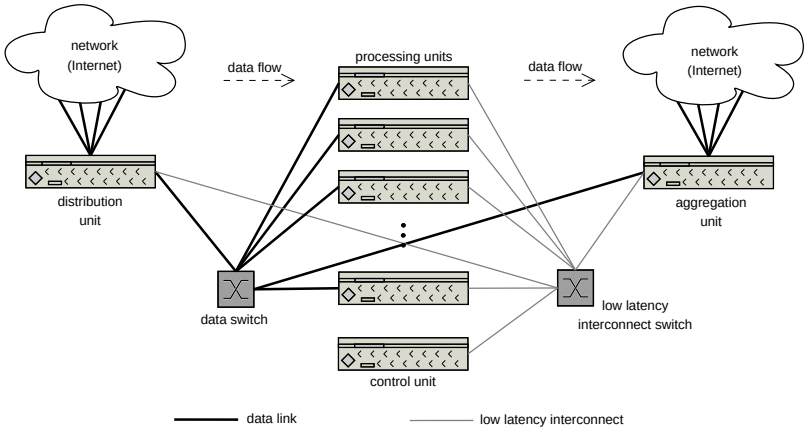


Fig. 1. Proposed DiProNN architecture.

The low latency interconnection is necessary since current common network interfaces like Gigabit Ethernet or 10 Gigabit Ethernet provide large bandwidth,

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

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 (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 desirable.

From the high-level perspective of operation, the incoming data are first received by the 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 node (or to the receiver). As obvious from the Figure 1, the DiProNN architecture comprises four major parts:

- *Distribution unit* takes care of ingress data flow forwarding to appropriate DiProNN Processing unit(s), which are determined by the Control unit described later.
- *Processing units* receive packets and forward 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* 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 are provided in Section 3) establishment and, if requested, providing feedback about their behavior.
- *Aggregation unit* aggregates the resulting traffic to the output network line(s).

2.1 DiProNN and Virtual Machines

Virtual machines (VMs) enhance the execution environment flexibility of the DiProNN node—they enable DiProNN users not only to upload the active programs, which run inside some virtual machine, but they are also allowed to upload the 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 allows strong isolation between virtual machines, and thus 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. 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. However, for our purposes this overhead is currently acceptable—at this stage we primarily focus on DiProNN programming flexibility.

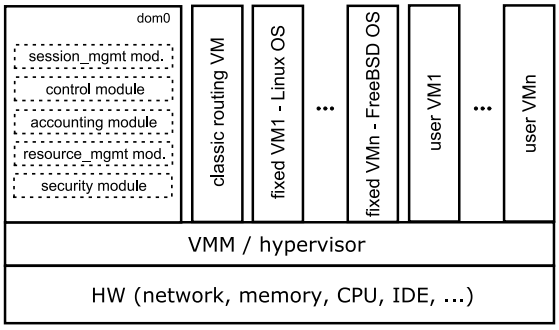


Fig. 2. DiPRoNN Processing Unit Architecture

2.2 DiProNN Processing Unit Architecture

The architecture of the DiProNN Processing unit is shown in Figure 2. The VM-host management system 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 other 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*[3] (ARTP, [1]) we designed. Depending on an application the user chooses the transmission protocol he wants to use—whether he wants or needs to use ARTP's extended functionality or not.

3 DiProNN Programming Model

In this section we depict a programming model we use for DiProNN programming. The model is based on the workflow principles [2] and is similar to the idea

[3] 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.

of the StreamIt [3], a language and a compiler specifically designed for modern stream programming.

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. 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 interfaces are reffered by their hierarchical names as shown in the example in Section 3.1.

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

There is one special attribute that can be set for active programs— *parallelizable*. If this attribute is set, the DiProNN performs the active program's processing in parallel[4]. 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 DiProNN usage).

3.1 DiProNN Program Example

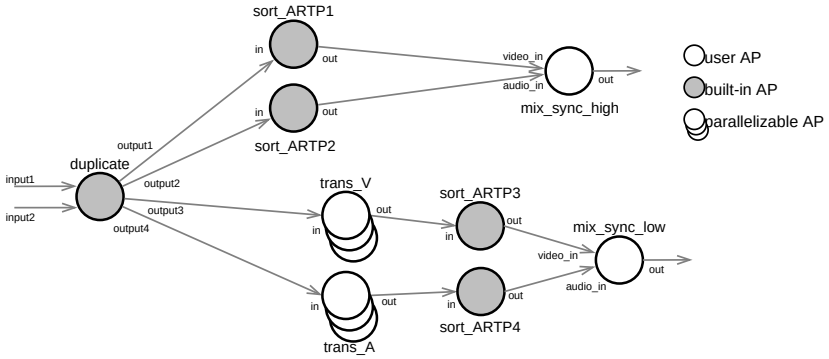
Let's have the following situation: there is one incoming high-bandwidth video stream (e.g., an HD stream having 1.5 Gbps) and one high-quality audio stream, both transferred using ARTP protocol described before. In the DiProNN, both streams must be transcoded into low quality streams for specified set of clients behind low-bandwidth lines, and for some clients the streams must remain in the original quality. At the output, there must be both audio and video streams of given quality mixed into just one output stream (thus having two output streams—one in high quality and one in lower quality) and the precise time synchronization between audio and video in both output streams is also required.

The possible DiProNN session graph together with the fragment of relevant DiProNN program is depicted in the Figure 3.

3.2 DiProNN Data Flow

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

[4] Note, that the DiProNN session must define on its own, how to distribute data over such parallel instances, or choose such distribution from built-in functions, e.g., round-robin or simple duplication principle.




```
Project first_project.HD_transcode;
# project parameters (owner, notifications,
# overall resource requirements, ...)
{AP name="sort_ARTP1" ref=localService.SortARTP;
  # AP parameters
  inputs = in;
  out = my_VM.mix_sync_high.video_in;
}
{VM  name="my_VM" ref=my_VM_image;
  # VM parameters
  {AP name="mix_sync_high" ref=mixer_syncer;
    inputs = video_in, audio_in;
    precision = 0.001;  # 1ms
    output = SEE_ARTP;
     # the real receiver inside ARTP packets
  }
  # other APs ...
}
# other VMs/APs ...
```

Fig. 3. Example of DiProNN session graph together with a fragment of relevant DiProNN program.

If the request could be satisfied, the session establishment takes place. It means, that the Control unit decides, which Processing units each active program will run on and appropriately sets both their control modules and the Distribution unit. Moreover, all the requested resources are reserved, if any.

Since the DiProNN programming model uses symbolic names for communication 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 (a part of each Processing unit) where the couple (symbolic name, real port) is using simple text protocol registered. The control module using this information together with the DiProNN program and,

if neccessary, the information set as an option in ARTP packet coming from an active program, properly sets the IP receiver of passing packets, which are then 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, DiProNN users may use the ARTP's extended functionality to make their active programs be 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 the one(s) given in DiProNN program.

The usage of symbolic names in DiProNN 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 at the control module of the Processing unit they run in.

4 Related Work

With the potential for many applications, the active networks principles are very popular and investigated by many research teams. Various architectures of active routers/nodes have been proposed—in this section we briefly describe only those mostly related to our work.

C&C Research Laboratories propose the CLARA (CLuster-based Active Router Architecture, [4]) providing customizing of media streams to the needs of their clients. The architecture of another programmable network node, LARA (Lancaster Active Router Architecture, [5]) in comparison with CLARA encompasses both hardware and software active router design. The LARA++ (Lancaster's 2^{nd}-generation Active Router Architecture, [6]), 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 Conclusions and Future Work

In this paper, we have proposed a VM-oriented distributed programmable network node architecture named DiProNN. The DiProNN users are able to arrange and upload a DiProNN session consisting of a set of their own active programs independent on each other and possibly running in their own virtual machine/operating system, and let their passing data being processed by the DiProNN. The communication among such active programs is provided using

standard network services together with 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 future challenges, the proposed DiProNN architecture is being implemented based on the Xen virtual machine monitor [7]. Further we 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 VMs (when they do not have their own VM specified), scheduling VMs 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). For the efficiency purposes we plan to implement some parts of the DiProNN in hardware, e.g., based on FPGA-based programmable hardware cards [8].

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

References

1. Rebok, T.: Active Router Communication Layer. Technical Report 11/2004, CESNET (2004)
2. Cichocki, A., Rusinkiewicz, M., Woelk, D.: Workflow and Process Automation: Concepts and Technology. Kluwer Academic Publishers, Norwell, MA, USA (1998)
3. Thies, W., Karczmarek, M., Amarasinghe, S.: Streamit: A language for streaming applications. In: International Conference on Compiler Construction, Grenoble, France (2002)
4. Welling, G., Ott, M., Mathur, S.: A cluster-based active router architecture. IEEE Micro **21**(1) (2001) 16–25
5. 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
6. Schmid, S.: LARA++ Design Specification (2000) Lancaster University DMRG Internal Report, MPG-00-03, January 2000.
7. 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)
8. Novotný, J., Fučík, O., Antoš, D.: Project of IPv6 Router with FPGA Hardware Accelerator. In: Field-Programmable Logic and Applications, 13th International Conference FPL 2003. Volume 2778., Springer Verlag (2003) 964–967