A Framework for Rapid Development and Portable Execution of Packet-Handling Applications

A Framework for Rapid Development and Portable Execution of Packet-Handling Applications

Mario Baldi and Fulvio Risso
Dipartimento di Automatica e Informatica
Politecnico di Torino
Torino, Italy
{mario.baldi,fulvio.risso}@polito.it

***Abstract* — This paper presents a framework that enables the execution of packet-handling applications (such as sniffers, firewalls, intrusion detectors, etc.) on different hardware platforms. This framework is centered on the NetVM — a novel, portable, and efficient virtual processor targeted for packet-based processing — and the NetPDL — a language dissociating applications from protocol specifications. In addition, a high-level programming language that enables rapid development of packet-based applications is presented.**

Keywords — High-speed packet processing, Network Virtual Machine, NetVM, NetPDL, NetPFL.

I. INTRODUCTION

Several network applications — such as packet routing, traffic classification, network address translation, packet sniffing, traffic analysis, traffic generation, firewalling, intrusion detection — perform some sort of packet processing. Some of them benefit from — or, in order to meet performance requirements, need — specific hardware support, such as offloading engines or application specific integrated circuits (ASICs), for some of their tasks. Even though packet processing is common to a large number of applications, at present no solution exists to delegate it to a single optimized component. This paper proposes a framework aiming at moving common packet-processing functionalities outside applications, while providing high processing efficiency.

Devising a general packet-processing component is not trivial since it should provide (*i*) a large number of packet handling functionalities executable with (*ii*) high performance on a (*iii*) wide range of hardware platforms. Virtual machines are the basis for *portability* and *flexibility*, although usually associated to poor performance. The proposed framework is centered on the *Network Virtual Machine* (*NetVM*) that aims at combining *high performance* with the common properties of virtual machines.

The issues to be addressed in the realization of such a framework, the motivations underlying such an effort, and the resulting potential benefits are discussed in Section II. Section III provides an overview of the proposed framework whose main component, the NetVM, is briefly described in Section IV. The Network Protocol Description Language (NetPDL) used to describe format and protocol encapsulation for packets handled by the NetVM is outlined in Section V. Programs developed in the NetVM framework can be written in some high-level programming language. Currently, one such language targeted at filtering applications has been defined and a prototypal compiler implemented; this language, called NetPFL, is presented in Section VI. Section VII discusses implementation alternatives for the NetVM, while Section VIII outlines deployment scenarios for the proposed framework, i.e., how it can be integrated into applications and how the NetVM interacts with the surrounding environment. Conclusions are drawn in Section IX.

This work has been carried out within the framework of the QUASAR project, funded by the Italian Ministry of Education, University and Research (MIUR) as part of the PRIN 2004 Funding Program. Its presentation has been supported by the European Union under the E-Next Project FP6-506869.

II. ISSUES, MOTIVATIONS, AND BENEFITS

One problem with a general packet-processing component stems from the *different flavors of packet processing* required by applications. For example, while an application might need to filter packets to further process only a subset of them, another one might need to modify the value of selected fields in each packet. A general packet processing component that fulfills the needs of all applications would be required to have a large set of functionalities, hence a high complexity. Another problem stems from the *high degree of portability* required. In fact, the above-mentioned applications need to execute in a specific location within the network (e.g., in the backbone, at the network edge, on end systems), in some cases, execution must be distributed across different devices. In general, such network applications must be deployed on very different (hardware and software) platforms, ranging from routers, to network appliances, to PCs, to smartphones. In some cases, the whole range of potential target platforms is not even known when application development takes place. Finally, most applications require *high performance* to cope with the large amount of packets flowing through high capacity links.

Virtual machines are the basis for the "write once, run anywhere" paradigm, thus enabling the realization and deployment of portable applications. However, general purpose virtual machines, such as Java and Microsoft's CLR, although widely used, do not achieve the needed performance when executing packet handling code. Instead, the NetVM framework has been designed specifically as a development and execution

platform for packet handling applications. Consequently, execution of packet handling related functions by the NetVM is optimized. Specifically, when the NetVM is deployed on network processors or hardware architectures packet handling related functions can be mapped directly on underlying special purpose hardware, e.g. ASICs, CAMs, etc.

Although the NetVM may have a more limited scope than Java and CLR virtual machines (i.e., the NetVM targets a smaller range of applications), its goals are somewhat more ambitious. In fact, the latter aim at application portability across platforms that, while different from hardware and software (i.e., operating system) point of view, are similar in being designed to support generic applications. Instead, the NetVM must combine portability and performance; this translates in the capability of effectively deploying available hardware resources (such as processing power, memory, functional units) notwithstanding the significantly different architecture and components of the various hardware platforms targeted.

The NetVM framework has the potential to bring the following contributions:

- simplify and speedup the development of optimized packet handling applications, such as traffic monitors, routers, firewalls because developers are no longer required to deal with low-level packet processing;
- functional modules can be developed and made available for reuse in new applications. This has the twofold advantage of (*i*) making development faster and (*ii*) applications more reliable and performing since they are based on stable and optimized modules;
- enable efficient mapping of the execution of software modules performing specific tasks onto optimized components of custom hardware architectures;
- provide a unifying programming environment and offer portability of packet handling applications across different hardware and software platforms;
- the NetVM architecture could be used as a reference design architecture for the implementation of hardware (integrated) packet handling systems;
- provide a new tool for specification, fast prototyping, and implementation of hardware (integrated) networking systems targeted to a specific packet handling application.

III. THE NETVM FRAMEWORK

The main objective of the framework presented in this work is to enable development of demanding, portable packet handling applications, i.e., applications that can be *efficiently* executed on *various hardware platforms*. The Network Virtual Machine (NetVM), the centerpiece of such framework, defines a new architecture for a (virtual) network processor in such a way that code execution exploits the high performance offered by network processors and ad-hoc hardware included in network systems, while also providing code portability and a reasonable degree of programmability. To this purpose the NetVM bytecode, called NetVM Intermediate Language (NetIL), includes packet handling specific instructions that virtualize ASICs and network processor functional units, hence enabling "easy" mapping on them.

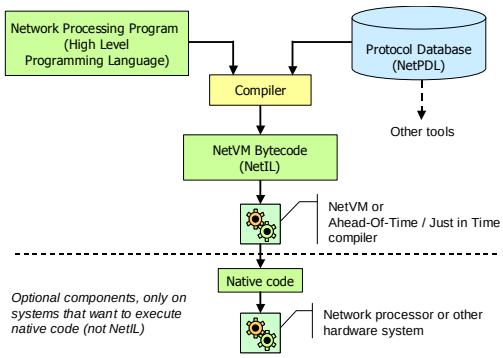

Figure 1 Architectural Vision.

Application developers are not required to write NetIL code as programs are generally written in a high-level programming language designed for networking applications, specifically for packet processing. NetPFL, one of these high-level programming languages, enables manipulations of packets and fields whose format is described through the Network Protocol Description Language (NetPDL) [2]. As shown in the top part of Figure 1, high-level code (e.g., NetPFL code) and the needed protocol definitions stored in a NetPDL database are compiled into NetIL code that can be executed by the NetVM.

NetIL code is translated into native code for the hardware architecture on which it is supposed to run. Although the NetVM can be implemented as an actual virtual machine, i.e., it can interpret NetIL instructions during execution, such a solution might not fulfill the performance requirements of many applications; therefore a NetIL compiler — possibly an ahead-of-time (AOT) or Just in Time (JIT) compiler — might be used to achieve maximum performance. Availability of virtual machine implementation or compilers for various hardware platforms guarantees portability; mapping of NetIL instructions on specific hardware components ensures high performance.

Since the NetVM framework makes implementation of packet handling functionalities simple and enables their efficient execution on multiple hardware platforms, it might become a reference architecture and a platform for supporting the development and execution of network applications. Packet handling code written with a high-level programming language and the needed packet descriptions are ultimately compiled into native code, which is able to run effectively on the target system. In particular, the NetVM framework could be especially beneficial for application development on network processors. In fact one of the problems hindering widespread deployment of network processors is their programming complexity. Each device offers its own programming environment that usually includes a C-like compiler, but the programming language is different not only from vendor to vendor, but even between different lines of products of a single vendor. This is due to the fact that C language is usually enriched with features that could help network application development (e.g. in order to enable exploiting some hardware characteristics of the network processor at best), while some features of the language (that are not

needed for packet processing) are disabled. Being this done on a platform by platform basis, portability is not guaranteed. Moreover, the most effective way to program these devices (while obtaining applications that efficiently run on them) still remains writing native assembly language, which is time-consuming, error-prone, and requires a deep knowledge of the target machine. Consequently, with either programming approach porting code from a platform to another one (even belonging to the same manufacturer) is a nightmare.

The NetVM framework offers a solution to these problems. Moreover, if proven to be effective, the NetVM could become a reference design for networking systems, as further elaborated in Section VII.

IV. THE NETVM

The main architectural choices of the NetVM were driven by *flexibility*, *simplicity*, and *efficiency* and leveraged on the experiences matured in the field of Network Processing Unit (NPU) architectures, which are specifically targeted to network packet processing. The resulting NetVM architecture is modular and built around the concept of *Processing Element* (NetPE), which virtualizes (or, it could be said, is inspired by) the micro-engine of a NPU.

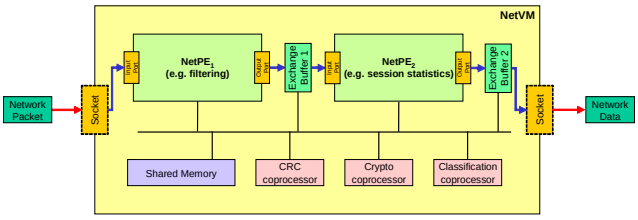

Figure 2 NetVM configuration example.

A NetPE is a virtual CPU — with its packet-processing specific instruction set and a local memory — that executes an assembly program that performs an individual function inside the NetVM and maintains a private state. A NetVM application is executed by several NetPEs (for example, Figure 2 shows an application deploying two NetPEs), each one dealing with only few tasks, but executing them efficiently. Complex structures can be built by interconnecting various NetPEs that exchange data through their *ports*. Moreover, as shown in Figure 2, NetPEs can use specialized functional units (coprocessors) and different types of memory to exchange data. This modular view derives from the observation that many packet-handling applications can be decomposed in simple blocks, thus possibly exploiting parallelism or pipelining to achieve better performance, since independent packets or processing blocks could be distributed to various NetPEs executing concurrently. The modular approach is not new: other software solutions, like *Netfilter* or *Click* have demonstrated its goodness and the parallel architecture of many network processors (based on multiple low-complexity microengines) follows the same principle.

Like most existing virtual processors, the NetVM, hence the NetPE, has a *stack-based design*. A stack-based virtual processor grants portability, a plain and compact instruction set and a simple virtual machine. The consequence of this choice is that no general-purpose registers are provided and the instructions that need to store or process a value make use of the stack. Each NetPE has its own stack.

The execution model is *event-based*, i.e. a NetPE is activated by external events, each one triggering a particular portion of code. Typical events are the arrival of a packet from an input port, the request of a packet on an output port or the expiration of a timer.

TABLE I NETIL INSTRUCTION CATEGORIES.

Category	Description	Example	Description
Initialization	Used to initialize the execution of a NetPE program	`set.share`	Set the size of the shared memory
Data transfer	Transfer data within memory	`dpcopy`	Copy a memory buffer from a portion of the memory to another
Pattern matching	Used to compare a value in a memory buffer against the top of the stack	`field.eq.8`	Compare the top of the stack with an 8bit field in memory
Flow Control	Used to control the execution flow	`jump`	Unconditional branch
Stack management	Used to manage the stack	`swap`	Swaps the two top elements of the stack
Aritmethig and Logic	Used to compute simple expressions	`ror`	Rotate right the value at the top of the stack

The NetVM Intermediate Language (NetIL) defines the NetPE instruction set that is similar to the one of a generic stack machine, with specific additions to support the particular architecture and application area of this virtual machine. Application specific instructions have been inspired to the instruction set of network processors. Opcodes can be subdivided into several groups; the most important ones are listed in TABLE I.

```
; Push Port Handler
segment .push

.locals 5
.maxstacksize 10
  pop               ; pop the "calling" port ID
  push 12           ; push the location of the ethertype
  upload.16         ; load the ethertype field
  push 2048         ; push 0x800 (=IP)
  jcmp.eq send_pkt  ; compare the 2 topmost values; jump if true
  ret               ; otherwise do nothing and return

send_pkt:
  pkt.send out1     ; send the packet to port out1
  ret               ; return
ends
```

Figure 3 NetIL code to filter IPv4 packets.

While more details on NetIL can be found in [1], Figure 3 shows a sample NetIL code that filters IPv4 packets encapsulated into Ethernet frames and returns them on port number 1 of the NetPE executing it. This NetIL program can be compared with an equivalent program (see Figure 4) for the Berkeley Packet Filter (BPF) [4], probably the best-known virtual machine in the packet processing arena. A first comparison highlights that the NetIL is definitely richer than the BPF assembly, which gives an insight about the possibility of the former. However, a NetIL program is far less compact: the “core”

of the IPv4 filter program in Figure 3 is six instructions compared to the tree of the BPF equivalent shown in Figure 4. This stems from a major NetVM architectural choice: a stack-based virtual machine is less efficient than a register-based virtual machine (such as the BPF). Anyway, an AOT/JIT compiler can overcome this limitation.

```
(0) ldh  [12]               ; load the ethertype field
(1) jeq  #0x800  jt 2  jf 3 ; if true, jump to (2), else to (3)
(2) ret  #1514              ; return the packet length
(3) ret  #0                 ; return false
```

Figure 4 BPF code to filter IPv4 packets.

V. NETPDL

Instructions of the high-level programming language of the NetVM framework operate on packet descriptions written with the *Network Protocol Description Language* (*NetPDL*) [2] and stored into a universal protocol description database — the *NetPDL database*. Having been designed with *simplicity* in mind, NetPDL is an application-independent language for the description of *packet header formats* and *protocol encapsulations* and does not support the description of a protocol temporal behavior — e.g., a protocol state machine. NetPDL is based on the eXtensible Markup Language (XML) that is becoming the preferred way for exchanging structured data between different applications. For this reason several tools, as both standalone programs and libraries, exist for dealing with XML documents and can be leveraged on.

NetPDL is the basis for enabling application independent, seamless packet processing because it offers a method to uniquely identify each protocol and each field. As a concrete example, among other things NetPDL provides a unique name (e.g. `ip.src`) for the entities dealt with by packet processing and usually referred to differently, e.g., "IP source address field", or "IP source address", or "ip source", or "IP.source". Moreover, the protocol header database can be dynamically changed to include new protocols or protocol features, without even restarting applications.

NetPDL was designed with the following objectives.

1. *Simplicity*: the syntax should be intuitive so that (1) it can be easily understood without a deep knowledge of the language and (2) protocol descriptions can be written using a simple text editor.
2. *Completeness*: the language must include a set of base primitives suitable to describe packet headers of the most common (present and possibly future) protocols, thus defining the way they can be processed. An external plug-in mechanism is provided for dealing with packet header formats that cannot be described by NetPDL primitives.
3. *Extensibility*: the language must support the addition of new primitives to the small set of base elements, allowing for the language to be tailored to a wide range of applications. Backward compatibility is ensured by applications being able to skip over unknown primitives.
4. *Efficiency*: the performance of applications deploying NetPDL databases must be comparable to the one of applications based on "hardwired" packet descriptions.

The above objectives have driven several choices in the XML-based specification of NetPDL. A sample of NetPDL is provided in Figure 5 that shows an excerpt of the Ethernet header description. Each primitive consists of an *element* characterized by several *attributes*. For instance, a header field is an element, the field size being an attribute of the element. Moreover, Figure 5 exemplifies that a NetPDL protocol description consists of two complementary parts. The *packet format* description contains the list and format of the fields constituting a packet header; the Ethernet header consists of 3 fixed-length fields[1], whose length is respectively 6, 6, and 2 bytes. The *protocol encapsulation* description, delimited by the `<encapsulation>` element, specifies how to determine the protocol (as indicated by the value of the `<protoref>` element) at the higher layer, i.e., how to interpret the sequence of bytes constituting the payload. In the sample in Figure 5, the identification of the higher layer protocol is based on the value of the `type` field, contained in the `switch` expression.

```
<protocol name="ethernet">
  <format>
    <fields>
      <field type="fixed" name="dst" size="6"/>
      <field type="fixed" name="src" size="6"/>
      <field type="fixed" name="type" size="2"/>
    </fields>
  <format>

  <encapsulation>
    <switch expr="type">
      <case value="0x0800"> <protoref name="IP"/>  </case>
      <case value="0x0806"> <protoref name="ARP"/> </case>
    </switch>
  </encapsulation>
</protocol>
```

Figure 5 Excerpt of the NetPDL description of the Ethernet Header.

VI. NETPFL

The Network Packet Filtering Language (NetPFL) is a simple high-level programming language targeted to generating network-oriented processing code for the NetVM. It supports a filter – action model: a filter is applied to the packet and, if this is satisfied, the corresponding action is taken. Current actions involve returning a packet on a specified port, classifying a packet (e.g. for calculating protocol distribution statistics), returning the content of a set of fields in the packet. Filters can be based on (*i*) protocols (i.e., the filter is satisfied by a packet containing a message of a given protocol) and (*ii*) field values (i.e. an expression involving one or more fields of one or more protocol headers). Filters are applied to the stream of packets entering the executing NetPE through its ports.

```
NetPFL:   ethernet.type == 0x800 ReturnPacket on port 1
tcpdump:  ether proto 0x800
```

Figure 6 High-level code to filter IPv4 packets, in NetPFL and tcpdump.

Figure 6 shows a sample NetPFL program and offers a glance of the (low) complexity of an application for the NetVM framework. The code in Figure 6 consists of a filter based on a field value and a traffic capture action. It instructs the NetVM

[1] For simplicity, Preamble, Start Frame Delimiter, and Frame Check Sequence are not shown in this sample description.

to return on its port number 1 all Ethernet frames that contain the value `0x0800` in their `ethernet.type` field. In other words, this code implements a filter for IPv4 packets.

For the sake of comparison, the same Figure shows the definition of the same filter with the widely known tcpdump [3] packet filtering application. The comparison shows that, even though the NetVM provides the flexibility of a generic packet-processing engine, defining a packet filter with NetPFL is not more complicated than specifying it for tcpdump, i.e., a utility specifically optimized for packet filtering. Hence, the increased flexibility of the NetVM is not traded for increased programming complexity, as well as for (significantly) lower performance, as discussed in the next section.

The NetPFL code in Figure 6 is translated into the NetIL bytecode shown in Figure 3 that can be interpreted by a NetVM implementation or compiled (through AOT/JIT) into native code on the target platform. Compiling the tcpdump filter within libpcap yields the BPF assembly code in Figure 4.

One of the biggest advantages of the NetPFL/NetPDL to NetIL compilation (shown in the top part of Figure 1) is that developers do not need to care about protocol encapsulations. For instance, when writing code to filter out any IPv6 packet, the compiler checks the NetPDL database for any possible IPv6 encapsulation (e.g. IPv6 in Ethernet, or IPv6 in VLAN in Ethernet, etc.). This offers an unprecedented degree of flexibility because encapsulation is managed through NetPDL files, which are automatically updated when the protocol database is replaced by a new one.

NetPFL is only an example of high-level language; however we envision the realization of more NetIL compilers, thus enabling the generation of NetVM code from the most common programming languages (e.g. C, possibly with some network-specific extensions).

VII. NetVM Implementation

The NetVM aims at providing programmers with an architectural reference, so that they can concentrate on what to do on packets, rather than how to do that. This has been dealt with once for all during the NetVM implementation. This section focuses on how to implement the NetVM on both end-systems and network nodes.

The first option is software emulation of the reference design: NetVM bytecode is interpreted and for each instruction a piece of native code is executed to perform the corresponding function. This option is applicable to any platform (ranging from personal computers to smartphones), it does not require specific hardware, but features the worst performance. Better performance can be achieved by leveraging on specific hardware, where available, to execute specific instructions of the NetVM bytecode. Being based on on-the-fly interpretation of NetIL instructions, both options possibly enable dynamic downloading of code, i.e., changes to an executing program.

A third implementation option that further improves performance consists in creating an AOT/JIT compiler, i.e. a tool that inspects NetIL code and translates it into device-specific code, as shown in the bottom part of Figure 1. For example, such a compiler can translate NetVM bytecode into x86 assembler (therefore making use of the processor registers instead of operating on a stack) or into code based on the instruction set of a specific network processor. A common objection is that a program written in C and compiled natively for a general-purpose target platform may run faster than the same program written in NetIL and transformed into native code by an AOT/JIT compiler. However, the situation may be different if the target platform is a NPU or a network specific hardware architecture, for which writing optimized native code might be difficult and cumbersome. Instead, AOT/JIT compilers may be very efficient in exploiting the hardware resources of the target machines. In particular, NetIL network-specific instructions can be efficiently mapped onto custom functional units (e.g., ASICs and FPGAs) that have been designed to optimally execute their tasks. Furthermore, writing programs in NetPFL has the additional advantage that packet processing code is further automatically optimized by that compiler, which implements not only compiler-related optimization techniques, but also application (i.e., packet processing) related optimizations.

A fourth option is to implement the NetVM architecture in hardware, i.e., the architecture of the virtual machine could be used as the basis for the design of a hardware architecture for network processing (e.g., a network processor). A VHDL program implementing the NetVM reference design can be used to create a new FPGA chip that implements the NetVM.

Taking this a step further, the NetVM code implementing a set of packet processing functionalities (e.g., a NetVM program that tracks the amount of IPv6 traffic) could be compiled in the hardware description of a hardware system that implements such functionalities. In other words, gates on an FPGA can be configured not to implement the NetVM environment, but to execute the specific program that has to be mapped on it. This solution trades some flexibility (running a new application requires reconfiguring the FPGA gates) for maximum efficiency.

In general, the NetVM framework can be effectively used for fast prototyping, specification, and implementation of network oriented hardware systems. Currently, only a prototypal NetVM implementation according to the aforementioned first option — i.e., a NetVM emulation — is available. This section presents a few numerical results as a first quantitative evaluation of the proposed architecture. TABLE II shows the time needed by the NetVM and the BPF to execute an IPv4 filtering program, i.e., the code shown in Figure 3 and Figure 4. As expected, the BPF outperforms the NetVM because, as mentioned before, the NetIL program is far less compact that the equivalent BPF assembly. However, the performance difference should be cancelled by an AOT compiler (currently available only for the BPF) translating virtual machine assembly into native code.

TABLE II Performance of NetVM Emulation.

Virtual Machine	CPU clock cycles for executing the “IPv4” filter
NetVM	392
BPF	64

Translating NetIL instructions into native code (through AOT/JIT compilation) tailored to the characteristics of the target platform according to the third aforementioned implementation option significantly improves performance. This justifies the choice of a stack-based machine, which is intrinsically slower, since its instructions are much simpler to be translated into native code. The implementation of a JIT compiler is part of our future work on the NetVM.

VIII. DEPLOYMENT

The NetVM can be considered a black box that performs some processing on data, more likely packets. In a possible configuration, an input socket accepts "raw" packets, while end results are sent out through an output socket. As a matter of fact, the NetVM does not have a preferred location for its execution in real systems. In other words, a NetVM can be implemented as a chip on a network interface or on a network device blade, as a kernel module in an operating system, as a helper module in a user-level application, etc. The NetVM is very flexible and fits in significantly different hardware and software architectures, provided that its input and output ports are properly connected to the rest of the system and the data format is compliant with NetVM rules. Currently, the NetVM has been implemented as a user-level module in the NetBee [5] library, but a porting to the kernel level is planned in order to increase the efficiency in the real-time processing of packets received though the network interface cards.

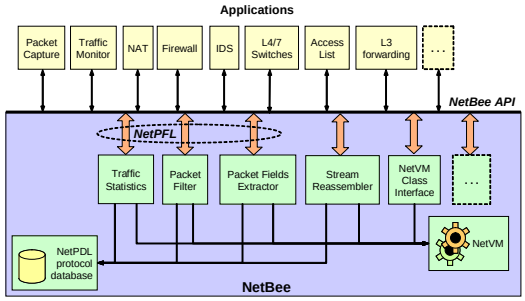

Figure 7 NetBee packet processing framework.

The NetVM is targeted to simple packet processing operations, although at very high speed. Therefore, the NetVM does not aim at supporting the development of complete applications (e.g. a stateful firewall); instead, a firewall can exploit the NetVM (and its framework) for its low-level processing. This vision is presented in Figure 7: the NetVM is a key component of the NetBee library, which implements a set of processing modules (e.g. a stream reassembly) that use the underlying processing capabilities of the NetVM.

A further step of deployment is depicted in Figure 8 that presents a scenario where all network devices have a built-in NetVM component, no matter how this is implemented. Developers can take advantage of having both the same instruction set and the same reference architecture on all the platforms, which constitutes a distributed processing environment where different network nodes execute NetVM programs coordinated by a remote console. Once NetVM support be provided by commonly deployed network gear, distributed applications could be based on downloading NetVM code on various network nodes (e.g. through some existing configuration protocols, such as SNMP) and possibly collecting the results yielded by its execution. This would open the path to new, powerful network-based applications.

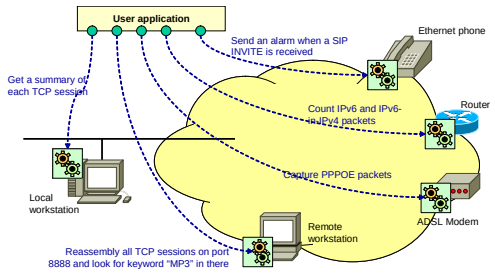

Figure 8 NetVM deployment scenario.

IX. CONCLUSIONS

This paper presents a framework for fast development of network applications. Its key components are the NetVM (Network Virtual Machine), targeted to the efficient execution of simple packet processing code, the NetPDL (Network Protocol Description Language), dissociating applications from protocol header formats, and a high-level language — such as the Network Packet Filtering Language (NetPFL) — for writing packet handling programs. The NetVM sets an architectural reference for program development; since it may be available on various hardware platforms and NetIL programs are portable, this lays the basis for distributed network processing.

The vision presented in this paper is currently under development in the open-source NetBee library, which is freely available on the web [5].

ACKNOWLEDGEMENTS

We would like to thank the several people who contributed to this project, among the others Loris Degioanni, Gianluca Varenni, Olivier Morandi, Francesco Andriani, Livio Torrero, Andrea Vesco, Leonardo Scucchia and Gianluca Dho.

REFERENCES

[1] L. Degioanni, M. Baldi, D. Buffa, F. Risso, F. Stirano, G. Varenni, "Network Virtual Machine (NetVM): A New Architecture for Efficient and Portable Network Applications," *8th IEEE International Conference on Telecommunications* (ConTEL 2005), Zagreb (Croatia), June 2005.

[2] F. Risso, M. Baldi, "NetPDL: An Extensible XML-based Language for Packet Header Description, to appear in "*Computer Networks* (COMNET), Elsevier.

[3] V. Jacobson, C. Leres and S. McCanne, libpcap, Lawrence Berkeley Laboratory, Berkeley, CA. Initial public release June 1994. Currently Available at http://www.tcpdump.org.

[4] S. McCanne and V. Jacobson, "The BSD Packet Filter: A New Architecture for User-level Packet Capture". Proceedings of the *1993 Winter USENIX Technical Conference*, San Diego, CA, Jan. 1993.

[5] Computer Networks Group (NetGroup) at Politecnico di Torino. The NetBee Library, available at http://www.nbee.org/, August 2004.