POLITECNICO DI TORINO
Repository ISTITUZIONALE

Energy transfer and energy level decay processes in Tm3+-doped tellurite glass

Original
Energy transfer and energy level decay processes in Tm3+-doped tellurite glass / Gomes, L.; Lousteau, Joris; Milanese, Daniel; Scarpignato, GERARDO CRISTIAN; Jackson, S. D.. - In: JOURNAL OF APPLIED PHYSICS. - ISSN 0021-8979. - ELETTRONICO. - 111:(2012), pp. 1-9. [10.1063/1.3694747]

Availability:
This version is available at: 11583/2497664 since:

Publisher:
American Institute of Physics

Published
DOI:10.1063/1.3694747

Terms of use:

This article is made available under terms and conditions as specified in the corresponding bibliographic description in the repository

Publisher copyright

(Article begins on next page)

07 May 2026

NETPDL: AN EXTENSIBLE XML-BASED LANGUAGE FOR PACKET HEADER DESCRIPTION

Fulvio Risso[1] and Mario Baldi
Dipartimento di Automatica e Informatica
Politecnico di Torino
Corso Duca degli Abruzzi, 24 - 10129 Torino (Italy)
{fulvio.risso,mario.baldi}@polito.it

Abstract—Although several applications need to know the format of network packets to perform their tasks, till now, each application uses its own packet description database. This paper addresses this problem by proposing the NetPDL, an XML-based language for describing packet headers, which has the potential of enabling the realization of a common, application-independent protocol description database that can be shared among several applications. Further, common functionalities related to the protocol database can be implemented in a library, which can be a basic building block for implementing networking applications.

Index Terms— Protocol Description Language, NetPDL, XML, Protocol Header Description.

I. INTRODUCTION

Several network applications — such as packet routing, traffic classification, network address translation, packet sniffing, traffic analysis, traffic generation, firewalling, intrusion detection — deal with packet headers, hence need to know packet formats. 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: packet processing is still implemented within applications by custom code.

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 those received, another one might need to modify the value of selected fields in each packet. A general packet processing component that fulfills the needs of any application 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 particular, the general packet processing component should be executable on a large number of platforms, ranging from hardware boxes (e.g. network switches), embedded devices (e.g. firewalls), and workstations, running a wide variety of operating systems.

The work was partly funded by Microsoft Research, Cambridge (UK) and the System On Chip Division of Telecom Italia Lab S.p.A., Torino (Italy).

[1] Contact information: Fulvio Risso, tel. +39 011 564.7008, fax +39 011 564.7099, e-mail fulvio.risso@polito.it.

These problems might have so far hampered the development of such a component. Irrespective of such problems, though, a first step towards moving packet processing functions out of applications consists in having a *universal protocol header database*, which is shared among all applications and contains packet descriptions for all network protocols. Current applications use a proprietary syntax to describe packet headers, and packet descriptions are often hardwired in their code. Consequently, supporting a new protocol requires the intervention of the developers of the specific application. Ethereal [4] and tcpdump [5], two well-known and widely deployed applications, have even two different protocol descriptions hardwired in their code: one used when filtering network packets in real-time, the other one when displaying packets in a user-friendly fashion. The first description is simple (and limited) because it is designed for high-speed operation (filtering). The second one is very comprehensive and the corresponding packet-processing engine is much slower than the one using the first description.

This paper presents the *Network Protocol Description Language* (*NetPDL*), an application-independent packet format description language that enables the creation of a universal protocol description database — the *NetPDL database*. One of the main design objectives of NetPDL, unlike alternative protocol description solutions (see Section III), is *simplicity*. For this reason, NetPDL is not intended as a protocol specification tool; for example, it does not support the description of a protocol temporal behavior — e.g., a protocol state machine. Instead, NetPDL is targeted to an effective description of *packet header formats* and *protocol encapsulations*.

NetPDL is based on the eXtensible Markup Language (XML) that is becoming the preferred way for exchanging structured data between different organizations. For this reason several tools, both stand-alone programs and libraries, exist for dealing with XML documents and can be leveraged for NetPDL handling. Moreover, XML documents are usually parsed by applications at run-time; by following the same approach with NetPDL, the protocol header database can be dynamically changed to include new protocols or protocol features, without even restarting applications.

Notice that NetPDL is beneficial also to applications for which a generic packet processing engine and a shared database are not cost effective. An example is provided by applications that perform simple operations on a small variety of packet headers. In these cases implementing packet processing within the application might be simpler than creating or interfacing a generic packet processing engine and leveraging from existing header descriptions might seem to bring negligible advantages. However, basing header processing code on NetPDL descriptions enables transparent (i.e., not requiring modifications to the application code) support of newer versions of the protocols.

Section II elaborates on the concept of various applications sharing a generic packet processing engine that operates according to protocol descriptions stored in a NetPDL database, where the latter can be dynamically updated. A survey of existing languages for describing network protocol headers is provided in Section III that highlights their limitations for the application context addressed in this work. Section IV presents an overview of the NetPDL language and the architectural choices behind it, while Section V provides the details of most primitives of the NetPDL language. Section VI gives an overview of NetPDL extensions, i.e. a set of additional primitives that can be used to enrich NetPDL for specific purposes. In particular, Section VI presents an extension aiming at the description of how packet information should be displayed. Finally, Section VII provides performance figures of a NetPDL-based engine implementation. Conclusive remarks are presented in Section VIII.

II. TOWARD NETPDL-BASED PACKET PROCESSING

Figure 1 depicts possible scenarios for the deployment of a common protocol description database shared by various applications. A packet processing engine, i.e., a *NetPDL-based engine* in this context, can be either embedded into each application (left-hand side of Figure 1) or shared among several applications.

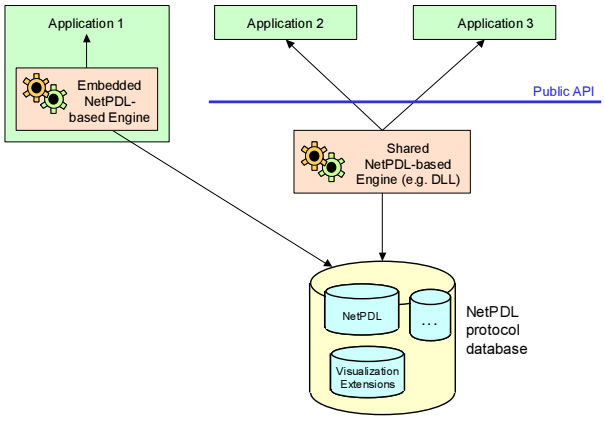

Figure 1. Relationships between applications, NetPDL protocol database, and NetPDL-based engines.

A NetPDL-based engine uses a NetPDL protocol database (*NetPDL database* in short), i.e. a set of XML files that contain a description of protocol headers, to learn the structure of the packets it is supposed to process. Since the NetPDL database is external to both applications and NetPDL-based engines, it can be updated without requiring modifications to the code implementing them. The NetPDL-based engine parses these XML files and creates an internal, engine-specific, representation of protocol headers. For example, based on the protocol description obtained from the NetPDL database, a (NetPDL-based) packet filtering engine can pre-calculate the offset of each field from the packet beginning. This will result in faster location of the requested fields within each incoming packet. Hence, a filtering engine based on the NetPDL language can have the same performance of one based on custom protocol descriptions, i.e., hardwired in the code of the filtering engine itself. Hence, performances of NetPDL-based engines do not depend on the characteristics of the language itself, which (being XML-based) may seem rather inefficient. From this point of view, a NetPDL description can be compared to a Java program, which can either be compiled into native code or interpreted at run-time. The execution time strongly depends on the tool used (compiler / interpreter), not on the language itself.

A NetPDL database can even be remotely stored on a centralized server accessible through the Internet. Geographically dispersed NetPDL-based engines can (periodically) download the most recent version of the NetPDL database, use the contained information to build their internal structures, and perform their processing. In this scenario NetPDL-based engines operate according to up-to-date and complete protocol descriptions contained in some external, remotely located XML files, while not suffering performance impairments during packet level processing.

This paper does not focus on any specific NetPDL-based engine, but rather on the definition of the NetPDL database. Nevertheless, Section VII provides measurements obtained with an existing NetPDL-based engine implemented in the NetBee library [3] with the objective of substantiating the above statements on performance.

Implementing protocol-processing engines based on an external description database has limitations when coming to second order optimizations. For example, since a NetPDL-based engine works on a per-protocol basis, optimizations that rely on the combined presence of two or more protocol headers cannot be implemented within a NetPDL-based engine. Finally, more investigation is required to assess the applicability and benefits of NetPDL-based packet processing in scenarios where performance requirements lead to the deployment of custom hardware optimized for a specific set of protocols. However, such application field is outside the scope of this work that focuses on software solutions.

III. RELATED WORK

This section provides an overview of known protocol description languages with specific emphasis on (1) support for packet header description, (2) support for protocol encapsulation description, (3) extensibility.

Libpcap [6], one of the most widely used packet processing libraries, provides a set of functions that allow to selectively capture packets by means of a filter. The filter, specified in high-level language (e.g. "*tcp*" means "*capture only TCP traffic*"), is translated into special assembly code that is executed by a filtering engine, the BPF (Berkeley Packet Filter) [6] virtual processor. Since the filter operates by checking the value of selected packet header fields, the protocol format must be known. Libpcap [7] embeds protocol definitions within its source code, which can be (hardly) modified only by recompiling the library. In essence, libpcap does not have a language to describe protocol headers; its simple language can be used to define a packet filter (operating on the most common protocol fields) and it cannot be used for other purposes.

One of the best-known protocol description languages is the one deployed by Analyzer 2.0 [1], a protocol analyzer developed at by one of the authors. An easily extensible C-like structure is used to describe packet header fields and protocol encapsulation. A most notable feature of the resulting packet processing architecture is the ability to both decode packets and customize their summary and detailed views based on external files — Description File Format (DFF) and Index File Format (IFF) configuration files. However, the Analyzer 2.0 protocol description language does not provide adequate support for variable-length fields and optional fields.

FALCON [9] — an evolution of the Analyzer 2.0 protocol description language, notwithstanding a different syntax — enables more complex computations, variable-length fields, and optional fields to be specified. However, FALCON provides only primitives for packet decoding (e.g. packet displaying is not taken into consideration) and its protocol description files have poor readability from the standpoint of a user working on them without any specialized (e.g. GUI) tool.

The GASP (Generator and Analyzer System for Protocol) Language [10] is similar to Analyzer's, but has a major emphasis on packet generation, rather than decoding. Like FALCON, it supports only header format description.

The protocol description language used by SPY [11] provides checking primitives to validate the correctness of selected fields and its protocol description files have excellent readability. However it does not provide proper support for optional fields. Like Analyzer 2.0, it supports some visualization primitives (albeit quite poor ones: only a detailed view of the packet is supported, with limited customizability); however this feature is natively provided by the language, while Analyzer does the same though an extension mechanism, which allows arbitrary future enhancements.

The protocol description language recently proposed in the JnetStream project [13] is probably the most flexible among the listed languages. It does support field format descriptions, optional fields (through conditional primitives such as *if-then-else* and more), field value validation and visualization directives (although the difference between *summary* and *detailed* view of the packet is not clear). However, it does not foresee extensions to the language, which means that new features not included in the "base" language cannot be added. In addition, the complete NPL (Network Protocol Language) description is not easy to read (despite its C-like syntax) since all directives are together without a clear separation between header format descriptions, field value validation and visualization directives.

The Solidum PAX Pattern Description Language [12] is targeted at pattern description, a pattern being either a set of fields or a set of protocols. For instance, the IP/Ethernet stack is considered the most common "pattern" to check when looking for ICMP packets. The language is designed with the objective of speeding up pattern matching operations on Solidum network processors. However, protocol encapsulation description with the PAX language is cumbersome since all the possible combinations of the formats for a given protocol have to be explicitly listed when describing the encapsulation of a higher layer protocol. Finally, PAX does not provide displaying and checking primitives and does not properly support optional fields.

Abstract Syntax Notation number One (ASN.1) [15] is an ISO standard notation often used to describe packet formats in protocol specification documents. ASN.1, although a standard, is not attractive for many applications, such as packet processing engines, due to its usage complexity. Moreover, several keywords are meaningless when dealing with protocol description, while other important features — such as support for describing protocol encapsulation — are missing. Furthermore, ASN.1 parsing is not trivial and only a few public and open-source tools do exist.

The ACT ONE language, part of the LOTOS standard [16], is another existing approach to header description. LOTOS, a Formal Description Technique standardized by ISO for the design of distributed systems, consists of two parts: (1) a process algebraic part is intended for modeling the dynamic behaviors of systems, and (2) a data algebraic part is proposed for modeling data structures and value expressions. While the former has a different purpose than NetPDL, the latter, based on the abstract data type language ACT ONE, compares to NetPDL. However, ACT ONE has been widely recognized to be complex to use: the very nature of its building elements and the rigidity of its model lead to lengthy specifications with a lot of repetitive, technical details that clutter the specification [17]. The enhanced version E-LOTOS [18], while addressing some of these issues, is still quite complex to use and heavily oriented to protocol specification and verification. Consequently, it does not support application-specific extensions, such as data visualization. With respect to the objectives and intended applications of NetPDL, analogous considerations apply also to Estelle [19], a currently withdrawn standard.

Finally, the ABNF notation [14] includes some interesting features that are useful when describing complex messages (like the ones of SMTP and HTTP); however it is fairly complex (due to its compact and very efficient syntax) and it does not include any extension mechanism for supporting other than header format descriptions.

In summary, the languages for protocol description proposed thus far display several weaknesses, especially with respect to extensibility, simplicity; most of them also lack effective support for optional fields. Particularly, none of the examined languages provides an explicit way to be extended and only some of them support the definition of visualization and checking primitives. These reasons motivated the creation of NetPDL, which aims specifically at addressing the above shortcomings.

IV. NETPDL OVERVIEW

This section presents the general architecture of NetPDL and the ideas behind the language. After a brief look at XML (on which NetPDL is based), an overview of the protocol description language will be presented.

A. XML Brief and Terminology

The eXtensible Markup Language (XML) [21] is a simple, flexible text format derived from the Standard Generalized Markup Language (SGML), a.k.a. ISO 8879 [22]. Originally designed to meet the challenges of large-scale electronic publishing, XML is playing an increasingly important role in a wide variety of data exchanges between computer systems using web-based protocols or others.

XML documents usually consist of *elements*, also called *tags*, delimited by the '<' and '>' characters. Each element contains a *name* and an optional set of *attributes*; a *value* might be optionally specified for each attribute. For instance,

```
<fruit name="apple"/>
```

is an element called `fruit`, with an attribute named `name` whose value is `apple`. Instead,

```
<person>John Black</person>
```

is an element called `person`, that does not have any attribute, and whose content is `John Black`. Elements can be nested (e.g. the content of `<person>` can include `<age>`) in order to create more complex structures.

The XML specification does not define elements and attributes. This is done by using the syntactical rules defined by XML to specify what elements, attributes and values are valid for a specific application. For instance, an XML-based language aimed at describing living creatures could include a tag called `<person>`, while a language aimed at describing food could include a tag called `<fruit>`. Text files compliant to either the XML DTD (Document Type Definition) or the more powerful XML Schema standards define the valid elements of a language. Although these files are not strictly compulsory, they provide a standard way to specify the syntactical rules of an XML-based language; therefore their use is strongly encouraged.

An application supposed to use XML-based documents must include an XML parser. The parsing process can be split into two steps. First, locating XML elements, attributes and their values; second, performing semantic actions associated to each element, e.g. create a new database record for the given person.

The first step is application-independent and several XML parsing tools, often called *XML engines*, are available, most notably the largely used Apache Xerces [24] and Microsoft's. For example, a typical output of the first parser is that the first valid XML tag within a document is the element `<person>`. The second step is, obviously, application-dependent. The creation of a NetPDL-based engine requires implementing only this second parsing step since an existing tool can be deployed for the first one.

B. Why XML

One of the most common objections moved to NetPDL concerns its being based on XML instead of a procedural language like C or Java. The main reason for having chosen XML is that tokens of traditional programming languages cannot be extended. For instance, although a `struct` in the `C` language may be suitable to describe a protocol header, different applications might need different kinds of information to be provided together with packet header formats. For example, an application might need a description of how a field should be printed (e.g. as a hex/dec number), while another might need a specification of the validity range of its values. Hence, an application programmer needs to be able to extend a packet description language to provide the constructs required to describe specific aspects needed by specific applications. Extensibility is one of the key features of NetPDL and can be easily obtained through the definition of new XML attributes associated to base language elements. The same result cannot be achieved with traditional programming languages unless heavily modified, hence changing their very essence. Instead XML has built-in *extensibility* due to the structure itself of an XML document based on elements and attributes (as described in the previous section). New elements and attributes can be added while preserving backward compatibility with prior application-dependent XML parsers that can simply ignore tokens that they are not able to process.

There are several additional reasons besides extensibility for choosing XML. Being plain text files, XML descriptions can be easily edited and debugged on any platform with a simple text editor. Overall, thanks to the availability of XML libraries that take care of the first parsing step, implementing parsers for XML-based documents is definitely simpler than for any other existing language, whose parser is usually based on the `lex` and `yacc` UNIX utilities. In addition, XML has the capability for strong syntactical validation, which may even be performed before the first parsing step by an application-independent XML tool according to companion definition documents following the DTD or XML Schema standards. This further simplifies the implementation and execution of application-dependent parsers since the validation process (which is performed automatically during the first parsing step) eliminates a large set of possible errors (e.g. invalid tags, out of range values, etc.). Last but not least, XML

offers *portability* across different platforms, specifically web-based ones. For instance, it is pretty simple to visualize an XML description as a web page following the formatting rules specified by a companion eXtensible Stylesheet Language (XSL) file.

C. NetPDL Basics

NetPDL aims at describing packets as defined by network protocol specifications. This includes two complementary descriptions:

- the **packet format**: the list and format of the fields constituting a packet, and
- the **protocol encapsulation**: the rules on the fields of a packet that determine how — i.e., according to which protocol — to interpret the sequence of bytes constituting the payload of the packet.

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. External plug-ins can be invoked by the NetPDL-based engine for dealing with packet header formats that cannot be described by the abovementioned 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 must be ensured when adding primitives: applications using the language must be able to skip over unknown primitives.
4. **Efficiency**: the performance of applications integrating or deploying (see the two possible architectures in Figure 1) NetPDL-based engines must be comparable to the ones of applications that include custom code for packet processing possibly based on hardwired packet descriptions.

The above objectives have driven several choices in the XML-based specification of NetPDL. 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.

```
<proto name="Ethernet">
  <fields>
    <fixed name="dst" size="6"/>
    <fixed name="src" size="6"/>
    <fixed name="type-length" size="2"/>
  </fields>

  <nextproto>
    <switch>
      <expr type="int">
        <fieldref name="type-length">
      </expr>

      <case value="2048"><protoref name="IP"/></case>
      <case value="2054"><protoref name="ARP"/></case>
    </switch>
  </nextproto>
</proto>
```

Figure 2. Excerpt of the NetPDL description of the Ethernet Header.

Figure 2 shows an excerpt of the NetPDL description of the Ethernet header that consists of 3 fixed-length fields[2], whose length is respectively six, six, and two bytes. As shown by this example, NetPDL represents each field as an element containing *n* bytes. The `<nextproto>` element contains the protocol encapsulation description, i.e., it specifies how to determine the protocol (as indicated by the value of the `<protoref>` element) following the current Ethernet header based on the value of the `type-length` field (as specified by the value of the `<fieldref>` element).

V. The Language

The main objective of the NetPDL specification is the description of packet header formats and network protocol encapsulation. This section briefly presents the elements and attributes used for these purposes.

A. General Structure

A NetPDL document, whose general structure is shown in Figure 3, contains elements that enable proper packet processing. The document consists of a set of descriptions, each one referred to a single protocol and contained into a `<proto>` element. Each description includes the elements specifying header formats (inside the `<fields>` element) and encapsulation (within the `<nextproto>` element), as shown in Figure 2.

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

```
<netpdl>

  <proto name="_startproto">
    <!- Determine which is the first header -->
    <!- that is present in the packet -->
  </proto>

  <proto name="FirstProto">
    <fields>
      <!-- field list -->
    </fields>

    <nextproto>
      <!-- encapsulation info -->
    </nextproto>
  </proto>

  <!-- Other protocols -->

  <proto name="_defaultproto">
    <!- "Last resort" protocol -->
  </proto>

</netpdl>
```

Figure 3. General structure of a NetPDL document.

A NetPDL-based engine will start processing a packet (e.g. its binary dump) by matching the byte sequence with the elements of the description in the order they appear.

Some predefined protocols (`_startproto` and `_defaultproto`) are used in special cases, such as the "first" protocol of the encapsulation sequence and the "last resort" protocol to be used when no suitable protocol description is available for processing the remaining data of a packet. More details will be presented in the next sections.

B. Base Elements for Packet Header Description

The headers defined by the majority of the protocols currently in use contain a small set of fields, which, most often, can be categorized according to one of the six types shown in Table 1.

NetPDL element	Description
`<fixed>`	Fixed-length fields, aligned to a byte boundary
`<masked>`	Field that contains bit fields
`<bit>`	Bit fields
`<variable>`	Variable-length field
`<line>`	CR/LF terminated variable length field
`<padding>`	Field realigning the header to a 16 or 32 bit boundary

Table 1 Basic field types defined in NetPDL.

The vast majority of header fields has a fixed length and is aligned to a byte boundary. Less frequently, a field is composed of a few bits or has a variable length that can be determined only at packet-processing time. Variable-length fields can be either *length bounded* (i.e., the length is specified by the value of another field) or *sentinel bounded* (i.e., a given character or string indicates the end of the field). The above types of fields can be specified through the `<fixed>` (for fixed-length fields), `<masked>` (identifying the part of a header that contains bit fields), `<bit>` (for a bit field), and `<variable>` elements. Additional characteristics, such as the length of a `<variable>` field, can be described by means of specific attributes. All the above types are deployed in the example shown in Figure 4.

Due to their widespread presence in packet headers, NetPDL includes two additional pre-defined field types: the *line field* (`<line>`) — an ASCII line, which is a string terminated by a carriage return (CR) or CR+LF (Line Feed) character — and the *padding field* (`<padding>`) — often used to align a protocol header to a 16 or 32 bit boundary.

A field is completely characterized by specifying its *length*, the number of *occurrences*, and its *position* in the packet. However, the latter two items are usually not needed, because normally a field is placed after a given preceding field and occurs only once. In order to keep the notation simple, only the length of the field (through the attribute `size`) must be always specified for fixed length fields. The description of fields repeated multiple times is addressed in Section V.C.4, while the position of a field can be specified through the optional attribute `offset`. A packet trailer is a typical case in which this attribute is deployed.

C. Advanced Elements for Packet Header Description

The elements described above are often not sufficient: the header of a protocol as common as IP is one such example. This section introduces a number of more sophisticated, yet generally useful elements.

1. Field Blocks

The `<block>` element, a container for fields, aims at improving readability of packet header descriptions. The content of a `<block>` element is included in a protocol description through the `<includeblk>` tag, whose functionality can be compared to the one of macro expansions encompassed by most high-level languages. When an `<includeblk>` tag is found, its content is replaced by the content of the corresponding `<block>` element.

We foresee two deployment scenarios for the `<block>` element. First, it can be used to isolate a portion of NetPDL code that has a distinctive identity or function. An example can be seen in Figure 4: each option of the IPv6 protocol is defined within a distinct block in order to organize the NetPDL code in a clearer way. The second scenario is related to compactness: the same block of fields can be present several times within the protocol; the `<block>` element allows defining it once and the `<includeblk>` element enables using it multiple times within the header stack or within different contexts. The OSPF protocol provides an example of the latter: a Link State Advertisement header can be found in several OSPF packets (e.g. database description packets, acknowledgements, and others). The group of fields is described once, and it is recalled several times by means of the `<includeblk>` element.

```
<proto name="IPv6">
  <fields>
    <masked name="ver-tc-flabel" size="4">
      <bit name="ver" mask="F0000000"/>
      <bit name="tos" mask="0F000000"/>
      <bit name="flabel" mask="00FFFFFF"/>
    </masked>
    <fixed name="plen" size="2"/>
    <fixed name="nexthdr" size="1"/>
    <fixed name="hop" size="1"/>
    <fixed name="src" size="16"/>
    <fixed name="dst" size="16"/>

    <loop type="while">
      <!-- Loop until we find a 'break' -->
      <expr type="bool">
        <number value="1"/>
      </expr>

      <switch>
        <expr type="int">
          <fieldref name="nexthdr">
        </expr>

        <case value="0">
          <includeblk name="HBH"/>
        </case>

        <!-- Other options follow here -->

        <default>
          <loopctrl type="break"/>
        </default>
      </switch>
    </loop>
  </fields>

  <nextproto>
    <switch>
      <expr type="int">
        <fieldref name="nexthdr">
      </expr>

      <case value="6"><protoref name="TCP"/></case>
      <case value="17"><protoref name="UDP"/></case>
    </switch>
  </nextproto>

  <block name="HBH" longname="Hop By Hop Option">
    <fields>
      <fixed name="nexthdr" size="1"/>
      <fixed name="helen" size="1"/>
      <variable name="Options" type="size">
        <expr type="int">
          <fieldref name="helen"/>
        </expr>
      </variable>
    </fields>
  </block>
</proto>
```

Figure 4. Extract from the IPv6 header description: `<loop>`, `<switch>`, `<block>` and `<includeblk>` elements.

2. Conditional elements

It is not uncommon within protocol headers that the presence (or a value) of a field depends on the value of another field. Optional fields, like IP options, are an example. Two NetPDL elements have been defined to address this issue: the `<switch>-<case>` and `<if>` elements. The former allows multiple alternative descriptions to be provided; the one actually considered when processing a packet is determined by the evaluation of a simple condition, which is usually dependent on the value of another field (the one named `"nexthdr"` in Figure 4). By default, each `<case>` compares a given value against the value of the field specified within the `<switch>` element, although some more complex conditions can be defined. For instance, the condition "*this `case` must be selected for all values between 10 and 20*" is given by `<case value="10" maxvalue="20">`. Conditions are evaluated in order; therefore a `<case>` description is applied only if no other preceding condition matches. The element `<default>`, also exemplified in Figure 4, indicates a "default choice" in case no other choice is suitable.

The `<if>` element enables the description of a group of fields to be made dependent on the evaluation of an arbitrarily complex condition, (e.g. the value of several other fields). The `<expr>` element can be used to define the condition in the context of the `<if>` element. An example of the `<if>` and `<expr>` element deployment can be found in Figure 5.

3. Expressions

NetPDL supports mathematical, logical and string expressions that are needed by conditional elements. Expressions are possibly the most complex structures of NetPDL, mainly because users are often used to think about expression in infix notation (`A+B*C`), which is not appropriate for XML. In fact, although some solutions for using such a notation exist even in XML, the choice for NetPDL has been to define expressions in "native" XML structures. This enables their syntactical correctness to be verified through simple XML rules.

Figure 5 includes a sample expression: the `<expr>` element defines the entire expression, while child elements define operands and operators.

4. Repeating a field or a group of fields

The repetition of a field (or a group of fields) is rather common and is addressed by the `<loop>` element. The following four variants identified by the `type` attribute as shown in Figure 4, are available.

- *Size-bounded loop*: a (group of) field(s) is iterated until the cumulative size becomes equal to a given value.
- *Occurrence-bounded loop*: a (group of) field(s) is iterated a given number of times.
- *While-bounded loop*: a (group of) field(s) is iterated until a given condition is true (see Figure 4).
- *Do-bounded loop*: a (group of) field(s) is present at least once; the number of repetitions depends on a given condition.

The `<loopctrl>` element forces a corresponding repetition (as specified by a `<loop>` element) to be restarted or interrupted — similarly to the C/C++ *break* and *continue* instructions. The description of the IPv6 protocol header, shown in Figure 4, contains a *while* variant of the `<loop>` element. An IPv6 header possibly consists of various optional headers; the number is not known in advance. Each optional header has a "next header" field (`<fixed name="nexthdr">`) that identifies the next optional header. Thus, a packet processing engine should continue looking for optional headers as long as the value of the "next header" field equals one of the values specified as valid. In Figure 4 this is expressed through a `<switch>`

element that lists all the valid values for the `nexthdr` field. The loop (i.e., repetition of the optional header) is repeated until an "invalid" value is found in the "next header" field[3]; in this case, the `<loopctrl>` element breaks the loop.

5. Expressions with Lookahead Operands

Sometimes, expressions are required to look at some bytes that have not been associated to any field yet. An example can be found in the widespread Ethernet and IEEE 802.3 headers: the field spanning the 13th and 14th bytes from the beginning of a frame can be either the Ethernet 2.0 `ethertype` or the IEEE 802.3 `length` field, depending on whether its value is greater than 1500 or not, respectively. Obviously, it is not possible to determine whether to interpret the above-mentioned pair of bytes as either a `length` or an `ethertype` field before actually checking their value. NetPDL addresses this situation through *lookahead operands*.

When evaluating an expression, a NetPDL-based engine can interpret the next bytes in a packet dump (which have not been assigned to any field) as they were belonging to a new field, and evaluate the expression according to this value.

```
<proto name="Ethernet">
  <fields>
    <fixed name="dst" size="6"/>
    <fixed name="src" size="6"/>

    <if>
      <expr type="bool">
        <lookahead>
          <fixed size="2"/>
        </lookahead>
        <oper="le"/>
        <number value="1500"/>
      </expr>

      <if-true>
        <fixed name="length" size="2"/>
      </if-true>

      <if-false>
        <fixed name="ethertype" size="2"/>
      </if-false>
    </if>
  </fields>
  ...
</proto>
```

Figure 5. Ethernet header description, differentiating between the IEEE and DIX formats.

The use of a `<lookahead>` operand for the description of Ethernet headers is shown in Figure 5, which can be compared to Figure 2 where the same header is described without differentiating between the IEEE and Ethernet 2.0 formats. The 13th and 14th bytes are evaluated as they were part of a `<fixed>` field, which is used to determine which branch of the `<if>` element specifies how to process the sequence of bytes. According to the selected branch, the two bytes used to evaluate the expression are interpreted as the actual field — `length` or `ethertype`.

6. Custom plug-ins

For the sake of simplicity, NetPDL does not aim at supporting the description of every possible feature ever defined within a protocol. Instead, NetPDL provides a `<plugin>` element to enable the deployment of external code to handle protocol header features not directly supported by NetPDL elements. The `<plugin>` element defines a link to custom code that can be part of either the NetPDL-based engine or an external library (e.g. a Dynamic Link Library in Win32). This element can be used, for example, to implement the processing of protocols that have a very complex structure (e.g. DNS, SNMP etc.).

Since a `<plugin>` is an interface toward special purpose native code, each NetPDL engine implementation must include (in addition to the code implementing NetPDL primitives) the ad-hoc code corresponding to the plug-ins possibly in use.

D. Protocol Encapsulation

This section presents the NetPDL primitives for handling protocol encapsulation, i.e., how to specify the protocol description to be used in processing a packet's payload. Protocol encapsulation is based on the `<nextproto>` element, as shown in the Ethernet description example in Figure 2.

For most protocols, encapsulation is based on the value of one or more header fields. The relationship between the value of such fields and the encapsulated protocol can be specified by the `<protoref>` element deployed within `<switch>-<case>` or `<if>` elements, which are used to evaluate the condition that brings to the correct encapsulated protocol. The `<switch>-<case>` element is usually deployed to identify an encapsulated protocol through the value of a single field (see Figure 2 for an example), while the `<if>` element allows encapsulation to be made dependent on complex conditions (see [20] for details and examples).

In some cases, the encapsulated protocol is not (univocally) identified by any field in the encapsulating packet header: further processing of the encapsulated header is needed to ensure proper identification of the corresponding protocol. For instance, a BOOTP (Boot Protocol) and a DHCP (Dynamic Host Configuration Protocol) packet can be discriminated only by the value of a field within their own header. The `<presentif>` element supports such cases by defining a condition to be evaluated *while* processing a packet header in order to verify that the correct header description is being used. If the condition is valued false, the protocol description used so far for the processing does not match the byte stream and another one must be selected. An example can be seen in Figure 6: the IPv4 protocol is characterized by the "version" field (`<fieldref name="ver"/>`) containing a value equal (`<oper type="eq">`) to *four* (`<number value="4"/>`). In case the portion of packet that would

[3] More precisely, the repetition is interrupted when the value of the "next header" field does not match any of the `<case>` elements. This happens, for example, when the next header is the TCP or UDP one.

correspond to that field contains a different value, the byte sequence being processed is not an IPv4 packet.

```
<proto name="IPv4">
  <presentif>
    <!-- Check that 'version' is equal to '4' -->
    <expr type="bool">
      <fieldref name="ver"/>
      <oper type="eq"/>
      <number value="4"/>
    </expr>
  </presentif>

  <fields>
    <masked name="verhlen">
      <bit name="ver" mask="F0"/>
      <bit name="hlen" mask="0F"/>
    </masked>
    ...
  </fields>
</proto>
```

Figure 6. Excerpt from the description of the IPv4 header: the `<presentif>` element.

Notwithstanding the versatility of its primitives, NetPDL does not support encapsulation rules for a few protocols. For example, there are no fields neither within the UDP nor the RTP (Real-Time Transport Protocol) headers that can be used to uniquely identify RTP packets. In particular, reliable identification of RTP packets requires processing of state information on an RTP session, which is not adequately supported by the current NetPDL specification. In fact, NetPDL provides only limited support, by means of custom variables (see Section E), for stateful packet processing.

Finally, a NetPDL-based engine takes pre-defined actions when the packet header format cannot be inferred by matching the provided NetPDL packet descriptions to the byte stream being processed. This happens at least in two cases. First, when starting processing a new data unit, the first bytes must be interpreted according to the link-layer type of the interface through which the packet dump has been received and cannot be inferred from the data itself. The `_startproto` primitive protocol identifier specifies according to which header description the first bytes of a sequence should be interpreted/processed. Second, when none of the available protocol descriptions can be mapped to the byte sequence being processed, the `_defaultproto` primitive protocol identifier specifies a sort of "last resort" protocol, i.e., a header description to be used when no other protocol description applies.

E. Variables

Variables can be declared within a NetPDL description and manipulated at run time. The validity of a variable can be limited in time — specifically, a *volatile* variable is valid only while processing one packet, while the content of a *static* variable is preserved through different packets — and in scope — specifically, a *local* variable is valid only when processing fields of the associated protocol header, while a *global* one is valid while processing the entire packet. Hence, a variable belongs to one of the four categories resulting from all combinations of validity: local–volatile, local–static, global–volatile, global–static. Permanent (*static*) variables allow information to be kept while processing subsequent packets, thus enabling a limited degree of stateful packet processing.

Each NetPDL-based engine has a number of predefined global variables containing common information such as the link-layer type and, length of the frame being processed and the total number of bytes available for processing (for example, the user can choose to capture and/or store only the first part of a packet), the number of bytes processed (e.g. decoded) within the frame, the number of bytes processed within the current protocol header, a timestamp containing the time at which the related byte sequence was captured on the network. These default variables can be accessed from NetPDL when processing the corresponding protocol headers. In addition, the user can create its own variables.

Short Ethernet frames provide an example of a situation in which global volatile variables are needed. The shortest Ethernet frame is 64 bytes, however the number of valid bytes in the payload can be smaller than 64 when carrying a short packet, hence "padding" is present. Since the Ethernet 2.0 frame does not have a "frame length" field, the number of valid bytes within the Ethernet payload is kept in a global variable and must be inferred when processing the next level protocol. According to the IP description in [20], a NetPDL-based engine processing an IP packet received within a short Ethernet frame will properly update the variable containing the Ethernet payload size.

VI. NETPDL EXTENSIONS

One of the most valuable characteristics of NetPDL is its *extensibility*, i.e. the possibility to add new keywords (that can be inserted as either attributes of existing NetPDL elements or new elements) that will be used by some applications for their purposes. An example of a possible extension is attaching to header fields information related to their validity range; for instance, some fields allow only a limited set of values, while others (e.g. CRC fields) must have a precise value.

Extending NetPDL in order to support new features is rather simple. As an example, this section presents the first extension to this language, called NetPDL Visualization Extension, which provides information on how a decoded packet should be displayed. For instance, a 32-bit string representing an IP address should be displayed in dotted-decimal form, while a 32-bit string representing a CRC should be displayed as a hexadecimal number.

Only the capability to parse protocol descriptions based on NetPDL core is required from a NetPDL-based engine. Besides this, a NetPDL engine might process only extensions relevant to the specific application for which it was designed (e.g., packet filtering). Therefore, a NetPDL-based engine that does not support new extensions simply ignores extension specific attributes and elements, thus operating on a description like the one in Figure 2.

Thanks to the properties of XML, extensions can be written in files separate from the "core" specification, improving both readability and maintainability (users can update files separately).

A. The NetPDL Visualization Extension

The NetPDL Visualization Extension (the complete specification can be found in [20]) has been designed to support protocol analyzers (a.k.a. sniffers), which need to display captured data streams in an intuitive and user-friendly way. While NetPDL elements provide protocol analyzers with enough information to decode packets, the Visualization Extension provides the additional information needed for displaying packet fields.

The existing NetPDL Visualization Extension allows the definition of two views: a *summary view*, which includes the most important fields to be shown for each packet, and a *detailed view*, which includes all the fields of each packet, in full detail.

Even though the NetPDL Visualization Extension does not represent the only possible solution for protocol visualization, it is noteworthy in terms of simplicity and efficiency. For instance, XSL Transformations [23] have been considered for the definition of the above views since our packet decoding engine (presented in Section VII) produces an XML output (the PDML format, described in [20]). Even though XSL Transformations allow for richer visualization features, the user has to learn yet another (rather complex) programming language in order to handle visualization, which is against one of the main goals motivating our work: simplicity. The NetPDL Visualization Extension, instead, defines only a few visualization primitives that are based on the same principles of the core primitives, hence quick to get familiar with. Moreover, experiences with XSL processing have shown it computation intensive and slow; particularly, processing time grows linearly with the number of protocols involved (although this might be just a feature of the software package we used). For instance, an experiment on a Pentium IV – 2.4 GHz machine has shown that displaying a packet requires about 2.5 ms with XSL (Xalan-C [25] implementation) and about 0.6 ms with the NetPDL Visualization Extension (more details are provided in Table 2). In any case, it is a decision of each NetPDL-based engine developer whether to implement the Visualization Extension or to rely on the XSL Transformations instead. For the sake of this paper the NetPDL Visualization Extension is just an example of a possible extension to the NetPDL language.

B. Displaying the details of a packet

The NetPDL Visualization Extension defines a set of elements and attributes that are used within a visualization section, or *template*, to specify how each field is to be displayed. Each NetPDL field contains a reference to a template (through the `showtemplate` attribute) that contains the relevant visualization elements.

The most important attributes contained in the visualization template are `showtype`, `showgrp`, and `showsep`, which determine respectively the format (hexadecimal, decimal, ascii, or binary) of each byte, how bytes must be grouped, and the separator string between the groups. For example the MAC source and destination fields in Figure 7 specify that the field should be shown using the `EthMAC` template. This template displays a field by splitting its value in two parts (of three bytes each, as specified by the `showgrp` attribute) of hexadecimal numbers (`showtype` attribute) separated by a "`-`" sign (`showsep` attribute). The final result looks like `000800-AB34F9`.

The `<showdtl>` element defines a custom template that can be used to describe more sophisticated displaying rules. For example, in Figure 7 the template associated to a MAC address (`EthMAC`) verifies the type of the address (i.e., unicast, multicast, or broadcast) and displays a string identifying such type.

Similarly, the `<showmap>` element compares the field value against a set of choices to determine a suitable displaying format. For example, in Figure 7, this mechanism is used to determine the manufacturer of the network interface card (which depends on the first three bytes of the MAC address) and to show its name. The `<showdtl>` and `<showmap>` elements use various tags, such as `<if>` and `<switch>` - `<case>`, already defined within the NetPDL.

In case the proper way of displaying specific information cannot be expressed by means of the previous attributes and elements, the `showplg` attribute references a custom visualization plugin natively implemented into the NetPDL-based engine. An example is a plug-in that displays the canonical name (e.g. `www.foo.bar`) corresponding to IP addresses. The plug-in mechanism in the Visualization Extension is similar to the one in the base language (`<plugin>` element in Section V.C.6) and enables full flexibility through complete customizability of packet visualization.

```
<proto name="Ethernet" longname="Ethernet 802.3">
  <fields>
    <fixed name="dst" longname="MAC Destination" size="6"
          showtemplate="EthMAC"/>
    <fixed name="src" longname="MAC Source" size="6"
          showtemplate="EthMAC"/>
    <fixed name="type-length" longname="Type - Length"
          size="2" showtemplate="FieldHex"/>
  </fields>
  ...
</proto>
...
<netpdlshow>
  <showtemplate name="FieldHex" showtype="hex"/>

  <showtemplate name="EthMAC"
                 showtype="hex" showgrp="3" showsep="-">
    <showmap>
      <expr type="string">
        <!-- Extract the first 3 bytes of the address -->
        <strfieldref startat="0" size="3"/>
      </expr>

      <case value="FFFFFF" show="Broadcast address"/>
      <case value="000001" show="SuperLAN-2U"/>
      ...
      <default show="code not available"/>
    </showmap>

    <showdtl>
      <pdmlfield attrib="show"/>
      <if>
        <!-- It extracts the first byte of the -->
        <!-- MAC address, then it matches the result -->
        <!-- against the 'xxxxxxx0' pattern -->
        <expr type="bool">
          <fieldref startat="0" size="1"/>
          <oper type="match"/>
          <pattern value="xxxxxxx0"/>
        </expr>

        <if-true>
          <text value=" Unicast address (Vendor "/>
          <pdmlfield attrib="showmap"/>
          <text value=")"/>
        </if-true>
      </if>
      ...
    </showdtl>
  </showtemplate>
</netpdlshow>
```

Figure 7. Complete NetPDL description, with visualization extensions.

C. Displaying the summary of a packet

The Visualization Extension includes a set of primitives for creating a summary view of each packet. A protocol summary should include the most important information to be displayed and how to display it; a NetPDL-based engine (particularly its visualization extension code) will put all the "protocol summaries" together transparently. The summaries related to all the protocols subsequently encapsulated into a packet are appended to one another to create a single string.

```
<proto name="Ethernet" showsumtemplate="eth">
  <fields>
    ...
  </fields>
  ...

  <netpdlshow>
    <showsumtemplate name="eth">
      <section name="next"/>
      <text value="Eth: "/>
      <pdmlfield name="src" attrib="show"/>
      <text value=" => "/>
      <pdmlfield name="dst" attrib="show"/>
    </showsumtemplate>
  </netpdlshow>
</proto>
```

Figure 8. NetPDL Visualization Extension: specifying the summary related to the Ethernet header.

As an example, Figure 8 shows the elements of the summary view for Ethernet protocol headers. Each Ethernet frame will be summarized with the string "`Eth:`" followed by the source MAC address, the string "`=>`" and the destination MAC address, i.e., a format looking like:

```
Eth: 0001C7-B75007 => 000629-393D7E
```

VII. Implementing a NetPDL-Based Engine

A first implementation of a NetPDL-based engine, also supporting the Visualization Extension, can be found in the NetBee library [3], which is currently used by the Analyzer 3.0 protocol analyzer [2]. Both tools have been released as open source software.

The NetPDL database shipped with Analyzer includes an experimental description of 64 protocols, mostly related to the TCP/IP suite, including Ethernet, Token Ring, VLAN, IP, IPv6, TCP, UDP, DHCP, DNS, RIP, OSPF, BGP, PIM. This NetPDL-based engine, implemented as a 500 Kbytes Dynamic Link Library (DLL) for Windows, decodes packets and generates detailed and summary views. Additionally, it can also perform packet filtering. Analyzer accesses NetPDL-related functionalities by invoking functions (such as `DecodePacket()`) exported by the DLL. The NetBee Library exports a very clean interface and decoding and printing a packet requires only a few lines of code as shown in Figure 9.

```
while (1)
{
struct _nbPDMLPacket *PDMLPacket;
struct _nbPDMLProto *ProtocolItem;

  // Read packet from file or network
  Res= PacketSource->Read(&PacketHeader, &PacketData);

  if (Res == nbFAILURE)
    break;

  // Decode packet
  Decoder->DecodePacket(DataLinkCode, PacketCounter,
                        PacketHeader, PacketData);

  // Get the current decoded packet
  PDMLReader->GetCurrentPacket(&PDMLPacket);

  // Print some global information about the packet
  printf("Packet number %d\n", PDMLPacket->Number);
  printf("Total lenght= %d\n", PDMLPacket->Length);

  // Retrieve the 1st protocol contained in the packet
  ProtocolItem= PDMLPacket->FirstProto;

  // Scan the current packet and print on screen the most
  // relevant data related to each proto contained in it
  while(ProtocolItem)
  {
    printf ("Protocol %s: size %d, offset %d\n",
             ProtocolItem->LongName, ProtocolItem->Size,
             ProtocolItem->Position);

    ProtocolItem= ProtocolItem->NextProto;
  }
}
```

Figure 9. Sample code using the NetBee library: decoding and printing the details of a packet.

Table 2 provides a comparison of the performance of NetBee's NetPDL-based engine vs. custom implementations. The test, run on a Pentium IV at 2.4 GHz, is based on the analysis of 5 traces, each containing at least 8000 packets captured on our University egress link. Traces contain complete packets and no filters have been set in the capture process. The analysis of each trace is repeated 10 times and the second best processing time is retained as representative of the task completion time and shown in Table 2. The execution time of the relevant code of a few well-known tools (Tethereal [4] for packet decoding, which is the no-GUI version of Ethereal and WinPcap [8] for packet filtering) is measured and compared to NetBee's. Both tools are able to decode packets in memory and, if required, to create a PDML file (XML-based format for packet detailed view) containing the packet description.

Table 2 shows that the performance of NetBee (which decodes protocols by means of a NetPDL-based engine) and Tethereal (which implements protocol dissectors with native code) is very similar, 75 µs and 66 µs, respectively. These numbers account only for the decoding time, since the decoded packet is only created in memory according to each tool internal format. In case only the most important information about each field is required (basically the field name, its position in the packet dump, and its size), NetBee further decreases the processing cost from 75 µs/packet to 39 µs/packet; this function, called "partial packet decoding" in Table 2 is not available in Tethereal. In case decoded packets have to be dumped on file (in PDML format), performance increase to 0.65 ms and 1.08 ms per packet for NetBee and Ethereal, respectively. However these increased figures are mostly due to the efficiency of the code that dumps results on disk, which does not depend on the decoding process.

Although these results provide only a general indication of the performance obtainable from NetPDL-based tools, they clearly demonstrate that the NetPDL language itself does not introduce performance penalizations; performance fully depends on the quality of the tool deploying the language, i.e., of the NetPDL engine implementation.

	Tool name	Results
Partial packet decoding	NetBee	39 µs/pkt
Complete packet decoding	Ethereal (native code)	66 µs/pkt
	NetBee	75 µs/pkt
Complete packet decoding + packet dump (PDML)	Ethereal (native code)	1077 µs/pkt
	NetBee	648 µs/pkt
Filter generation for packet filtering	WinPcap (native code)	<1 ms
	NetBee	2831 ms

Table 2. Native code vs. NetPDL-based engine implementation performance.

The NetBee library includes also a second set of classes that implement packet filtering capabilities. These classes compile a high-level filter (in a language similar to the libpcap one) to pseudo-assembly instructions, which are executed by a virtual machine for processing packets. The NetPDL language is therefore used for describing packet formats and encapsulations which are intended to generate filters, while packet filtering itself is outside the scope of the language. Although the filter generation times shown in Table 2 are quite long, which is at the moment due to the very immature stage of the compiler, the implementation of these functionalities is important to demonstrate how a single NetPDL database is successfully and effectively deployed by different applications.

VIII. CONCLUSIONS

This paper presents the *Network Protocol Description Language* (NetPDL), a new *extensible*, *XML-based* language for describing the format of protocol headers. Network applications can base their packet processing on the NetPDL protocol database, i.e., a collection of packet descriptions. According to this architecture, new protocols can be easily supported by updating the protocol database instead of changing the application. For instance, tools like Ethereal and libpcap/WinPcap must be extended (i.e. new code is to be added and the whole tool recompiled) in order to support new protocols, while Analyzer needs only to update its protocol database without any recompilation. Simplicity in upgrading makes NetPDL advantageous also for applications that deal with a small set of protocols, for which an external protocol database might seem to offer no advantage.

In any case, this is not the only way that NetPDL can be beneficial for packet processing. As many other technologies (XDR, ASN.1, IDL, …), the NetPDL protocol database can be used to generate C code implementing packet processing. The target application includes the dynamically generated code as part of its sources, thus being able to run this code (natively) at very high speed. In this scenario upgrading applications requires automatically generating new source files out of an updated NetPDL database and re-linking.

The paper presents the basic NetPDL primitives needed for packet format description. NetPDL can be easily *extended* to satisfy the needs of specific applications, its syntax is *easy to understand*, and implementing parsers and support tools (such as graphic editors) is particularly simple thanks to the large number of existing XML-support tools and libraries. NetPDL further allows extensions aimed at more specific functions; as an example, this paper describes the *Visualization Extension* that provides primitives for describing how packets should be displayed, in a detailed and summarized view. NetPDL has been developed as a first step in an effort to relieve network applications from tasks related to packet processing. The basic idea is to delegate such tasks to a packet-processing engine that operates based on "standard" packet descriptions. NetPDL is intended as the formalism for such packet descriptions that can be gathered in an application independent *NetPDL protocol database*.

This will result in shorter development time for network applications and tools like protocol analyzers, firewalls, network monitors. NetPDL provides a general way to describe the format of protocol headers, *eliminating the need for implementing a custom protocol database in each tool*. This is a major change in the network tools arena, where each application currently defines its own protocol database that is often "hardwired" into the application code.

NetPDL-based engines, that perform packet processing based on NetPDL descriptions, can be implemented ad hoc for specific applications. In addition, if this approach became popular, it could trigger the creation of a set of "general purpose" NetPDL-based engines, which can be delegated by the applications to perform low-level tasks like packet decoding, packet filtering, and field extraction. Applications can include a generic NetPDL-based engine as external library or interact with an external NetPDL-based engine through a public API (Figure 1). The availability of off-the-shelf NetPDL-based engines will *speed up the development of network applications* because programmers can concentrate on high-level tasks (e.g. implementing filtering rules into a firewall) instead of dealing with low-level issues (i.e. how to locate a certain field in a given protocol header). Moreover, given their wide applicability, the above-mentioned NetPDL-based engines can be extensively tested, deeply debugged, and optimized; consequently, their deployment will result in *improved application quality*.

The Analyzer 3.0 network sniffer is based on a NetPDL-based engine (implemented within the NetBee library) that uses both the basic NetPDL language and the Visualization Extension. The NetBee Library has been made freely available with a BSD-style license in order to encourage the adoption of the presented technologies by both academia and industry. Complete specifications of the NetPDL language and the related technologies are available at [20].

Acknowledgements

The authors wish to thank Leonardo Scucchia, who laid the basis for this work during his Laurea graduation project and Francesco Andriani, who contributed to the first implementation of a NetPDL-based engine.

BIBLIOGRAPHY

[1] Computer Networks Group (NetGroup) at Politecnico di Torino, *Analyzer*, available at http://analyzer.polito.it, March 1999.

[2] Computer Networks Group (NetGroup) at Politecnico di Torino, *Analyzer 3.0*, available at http://analyzer.polito.it/30alpha/, March 2003.

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

[4] Ethereal, a public-domain sniffer. Available at http://www.ethereal.com.

[5] Tcpdump, a public domain sniffer. Available at http://www.tcpdump.org.

[6] S. McCanne, 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), USENIX.

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

[8] Computer Networks Group (NetGroup) at Politecnico di Torino, *WinPcap Web Site*, available at http://www.winpcap.org, April 2003.

[9] Surasak Sanguanpong and Ekapol Rojratanavichai, *Syntax Directed, Definition Supported Universal Protocol Analyzer*, Electrical Engineering Conference (EECON), Kasetsart University, Bangkok, December 1999. Available at http://anreg.cpe.ku.ac.th/pub/protocol.pdf (in Thai).

[10] Laurent Riesterer, *Generator and Analyzer System for Protocols* (GASP), March 2000, Available at http://laurent.riesterer.free.fr/gasp/.

[11] Christian Lorenz, *SPY LAN Protocol Analyzer*, 1999, available at http://www.gromeck.de/Spy/.

[12] Solidum (now Integrated Device Technology - IDT), *PAX Pattern Description Language*, October 2002, available at http://www.solidum.com/products/pax_pdl.cfm.

[13] Mark Bednarczyk, *JnetStrem Project*, http://netrepository.net.

[14] D. Crocker, *Augmented BNF for Syntax Specifications: ABNF*, RFC 2234, IETF Network Working Group, Nov. 1997

[15] Olivier Dubuisson, *ASN.1 - Communication Between Heterogeneous Systems*, Morgan Kaufmann Editor, October 2000.

[16] International Organization for Standardization. Information Processing Systems - Open Systems Interconnections - *LOTOS - A Formal Description Technique Based on the Temporal Ordering of Observational Behaviour*, Standard ISO 8807, 1989.

[17] C. Pecheur, *VLib: Infinite virtual libraries for LOTOS*, proceedings of the IFIP TC6/WG6.1 13th International Symposium on Protocol Specification, Testing and Verification, 25-28 May 1993, Liège, Belgium. Available at ftp://ftp.run.montefiore.ulg.ac.be/pub/RUN-PP93-03.ps.

[18] International Organization for Standardization. Information technology - *Enhancements to LOTOS (E-LOTOS)*. Standard ISO/IEC 15437:2001, 2001.

[19] International Organization for Standardization. Information processing systems - Open systems Interconnection - *Estelle - A formal description technique based on an extended state transition model*. Standard ISO/IEC 9074, 1997.

[20] Computer Networks Group (NetGroup) at Politecnico di Torino, *The NetPDL Specification*, May 2003, available at http://www.nbee.org/NetPDL/.

[21] Tim Bray, Jean Paoli, C. M. Sperberg-McQueen, Eve Maler, *Extensible Markup Language (XML) 1.0 (Second Edition)*, W3C Recommendation, 6 October 2000.

[22] International Organization for Standardization. Information Processing — *Text and Office Systems - Standard Generalized Markup Language (SGML)*, ISO 8879, 1986.

[23] James Clark, *XSL Transformations (XSLT) Version 1.0*, W3C Recommendation, 16 November 1999.

[24] The Apache Project, *Xerces: XML Parsers in Java and C++*, available at http://xml.apache.org/.

[25] The Apache Project, *Xalan: XSL stylesheet processors in Java and C++*, available at http://xml.apache.org/.