Using XML for Efficient and Modular Packet Processing

Original

Availability:
This version is available at: 11583/1494644 since:

Publisher:
IEEE

Published
DOI:

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)
Using XML for Efficient and Modular Packet Processing

M. Baldi and F. Risso
Politecnico di Torino, Dipartimento di Automatica e Informatica
Corso Duca degli Abruzzi 24, 10129 Torino, Italy
{mario.baldi, fulvio.risso}@polito.it

Abstract—XML is a technology that has been widely adopted for
data exchange, particularly in web and e-commerce applications.
This paper proposes the use of XML also for network packet
processing. It presents some XML-based languages for data
exchange and it identifies some examples in which XML can
enable a new, modular design of network applications while
maintaining the required high processing efficiency. These
technologies have been implemented in the NetBee library, which
provides an excellent way to give an insight of the performance
obtainable with the proposed approach.

Keywords: Network Packet Processing, Modular Packet
Processing, XML, NetPDL, PDML, PSML.

I. INTRODUCTION

While in the past applications used to work mostly alone, the
Internet has changed this paradigm and nowadays applications
are day after day more network-centric. In this respect,
applications tend to find more convenient focusing on a
specific problem, delegating other secondary tasks (which can
be seen as “non-mission critical”) to other applications. The
increased modularity of the applications brings to high
processing efficiency (everyone does what it knows best) and it
is possible thanks to the extremely fast and efficient data
transfers provided by nowadays networks.

This modular approach currently does not exist in network
applications. For instance, we can envision two categories of
network applications. The first category includes applications
(such as a web client and a web server) that use the network as
a simple “pipe” for transferring data. These applications
send/receive data through some form of high-level interface
(e.g. TCP/IP sockets) and are not interested in the internals of
the network itself. The second category (the one we are
interested in) includes applications that have to deal directly
with network packets and must have a deep knowledge of
network mechanisms. As examples, we can cite firewalls,
network address translators, intrusion detectors, packet sniffers,
network monitors, and more.

Modular processing can bring a valuable advantage to the
latter type of applications because it avoids wasting resources
in dealing with low-level details instead of concentrating on
their “core business”. For instance, a company that creates a
firewall should concentrate its efforts (e.g. lines of code and
time of its programmers) in checking whether a packet contains
malicious code instead of spending resources in locating the
TCP payload. This should be delegated to an external (and,
hopefully, optimized) component. This currently does not
happen and, right now packet processing is still implemented
within the application by application-specific code.

A clear example of the benefits of modular processing can
be seen in the next figures. Figure 1 shows a fragment of code
that checks if an Ethernet packet contains a TCP payload. In
this first example, the code checks if the Ethernet frame
contains an IPv4 packet, and then if the IPv4 packet contains a
TCP payload.

```
if (packet[12]==0x800 && packet[23]==6)
/* TCP packet */
else
/* Non TCP packet */
```

Figure 1. Filtering TCP packets on Ethernet/IP.

In case the application wants to support also IPv6, the code
must be modified as shown in Figure 2 in order to take into
account also the additional possible encapsulation.

```
if (((packet[12]==0x800) && (packet[23]==6)) ||
(packet[12]==0x86dd) && (packet[20]==6)))
/* TCP packet */
else
/* Non TCP packet */
```

Figure 2. Filtering TCP packets on Ethernet/IPv4-IPv6.

Obviously, this code may become a nightmare if the packet
can use any possible link layer (Ethernet, etc) or if the IPv4-6
headers have optional parts because of the very large number of
controls needed to locate if the packet contains a TCP payload.

This example demonstrates how packet processing can be
complicated, prone to errors, and it can be of little interest for
programmers that want to perform some high-level processing
to the packet. They should be very happy to write a fragment of
code like the one shown in Figure 3, in which the low-level
packet processing is delegated to another entity, such as an
external library.

```
if (Packet.Contains("tcp"))
/* TCP packet */
else
/* Non TCP packet */
```

Figure 3. Fragment of code that relies on some external module to filter
packets containing a TCP payload.
An example of modularity applied to packet processing can be seen in Figure 4. A very large set of applications can take advantages from a set of optimized components (e.g. a packet decoder or a packet filter), implemented in external modules.

Although NetPDL, PDML and PSML are only the first examples of XML-derived languages deployed in packet processing, they proved to be extremely useful albeit still efficient. These languages can be seen as the first step for creating a set of components that will enable modular packet processing, as shown in Figure 4.

Particularly, NetPDL is a key component for modular packet processing. NetPDL is a language that can be used not only as standard data exchange format, but it can enable the creation of protocol-independent applications. In fact, current applications use a proprietary syntax to describe packet headers; moreover, packet descriptions are often hardwired in their code. Consequently, supporting a new protocol requires the intervention of the developers of the specific application. Some well-known and widely deployed applications, like tcpdump [3] and Ethereal [4], have even two different protocol descriptions hardwired in their code: one used when filtering network packets in real-time, the other one for displaying packets in a user-friendly fashion. The first description is simple (and limited) because it is designed for high-speed operations (filtering). The second one is very comprehensive and the corresponding packet-processing engine is much slower than the one using the first description. NetPDL allows creating applications in a protocol-independent way without losing in efficiency, because application can read a generic protocol description according to the NetPDL language and will automatically be able to process packets containing that protocol.

This paper is structured as follows. Section II presents the NetPDL language for describing protocol headers, while Section III describes the other two XML-based languages for data exchange, PDML and PSML. Section IV briefly presents some of the characteristics of the NetBee library, which is a first example of library for modular packet processing. Particularly, it implements a packet decoder module (i.e. it receives the hexadecimal dump of a network packet and decodes it) using the proposed technologies. Finally, some conclusive remarks are presented in Section V.

II. NetPDL: Describing the Packet Format

The Network Protocol Description Language (NetPDL) [13] is a simple, application-independent packet format description language that is targeted to an effective description of packet header format and protocol encapsulation. For instance, NetPDL is not a protocol specification tool and it does not support the description of a protocol temporal behavior — e.g., a protocol state machine.

Some efforts have been done in the past to create a language that aims at describing protocol headers [6] [7] [8] [9] [10] [11] [12]. However, these approaches are usually very limited in their objectives, often poorly supported, and with poor performances. NetPDL aims at being a very simple language, which can be easily extended thanks to its XML-based structure.

A. The NetPDL Language

Each primitive consists of an element characterized by several attributes. For instance, a header field is an element, the field length being an attribute of the element.
Figure 5 shows an excerpt of the NetPDL description of an Ethernet header. Such a header consists of 3 fixed-length fields, 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` attribute). 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.

![Figure 5. Excerpt of the NetPDL description of an Ethernet Frame.](image)

The headers defined by the majority of the protocols currently in use contain a set of fields, which, most often, can be categorized under six different types. The vast majority of header fields has a fixed length and is aligned to a byte boundary, hence the `<fixed>` element. Less frequently, a field is composed of a few spare bits, hence `<masked>` (identifying the part of a header that contains bit fields) and `<bit>` (for a bit field) elements are defined. Other fields, are characterized by the fact that the length can be determined only at packet-processing time. These variable-length fields can be either `length-specified` (i.e., the length is specified by the value of another field) or `sentinel specified` (i.e., a given character or string indicates the end of the field). For them, the `<variable>` element exists.

Due to their widespread presence in packet headers, NetPDL includes two additional pre-defined field types: the `line field` `<line>` — an ASCII string terminated by a carriage return character — and the `padding field` `<padding>` — often used to realign the protocol headers to a 16 or 32 bit boundary.

Although a field is completely characterized by specifying its `length`, the `number of occurrences`, and its `position` in the packet, the latter two items are usually not needed because a field B is usually placed after its preceding field A and the number of occurrences is usually one. In order to keep the notation simple, only the length of the field (through the attribute `size`) must be always specified. The language addresses also the description of fields repeated multiple times, 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.

B. Advanced NetPDL Elements

Elements defined previously are often not sufficient; for example, the header of a protocol as common as IP cannot be described through the already presented elements. NetPDL defines also more sophisticated elements for conditional decoding (e.g., a protocol may have some optional headers, which may be present depending on the value of some fields), field loops (a field may be repeated several times depending on some condition) and storage support (a protocol may need to store some information for later processing). Finally, an element that recalls a custom plug-in can be defined for the cases in which no suitable NetPDL elements are available and the processing must be done through ad-hoc (native) code. For example, a well-known protocol such as the DNS contains a set of structures that aim at saving spaces within the packet by storing a pointer to a name instead of the complete DNS name. Defining a NetPDL element that implements this kind of processing does not seem suitable because no other protocols rely on this mechanism; hence, custom plug-ins provide an easier solution. Plug-ins allow processing the packet without increasing the complexity of the NetPDL language, i.e. avoiding the definition of new elements of little validity.

C. NetPDL extensions

One of the main 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. A NetPDL-based engine is required to parse protocol descriptions based on NetPDL, while it might process only the extensions relevant to the specific application for which the engine 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 5.

An example of a possible extension is the information related to the validity of each field; for instance, some fields allow only a limited set of values, while others (e.g. CRC fields) must have a precise value. Currently, the first extension to this language (called NetPDL Visualization Extension [13]) provides information on how a decoded packet should be displayed. For instance, a 32 bit number representing an IP address should be displayed in dotted-decimal form, while a 32 bit number representing a CRC should be displayed as a hexadecimal number.

The 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. These extensions defines a set of elements and attributes that are used within a visualization template. Each protocol and protocol fields can contain a link to the proper visualization template through the attributes `showtemplate` and `showsumtemplate`, as shown in Figure 6.

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, fields MAC Source and MAC Destination in Figure 6 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. Figure 6 show also the visualization template applied to the whole protocol in order to create the summary view. Each Ethernet frame will be summarized with the string “Eth:” followed by the source MAC address, the string “=" and the destination MAC address, producing a string looking like:

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

D. Performance evaluation

Most of the critics to the NetPDL language focus on its supposed performance penalty against a tool that contains the protocol definition hardwired in its code. Therefore, we run some test and we compared the Packet Decoder module of the NetBee library [2], which is a first implementation of a packet decoder entirely based on the NetPDL language, against the Tetheriel [4] packet sniffer, which is the no-GUI version of the well-known Ethereal. Tests, executed on a P4 - 2.4 GHz PC, are based on the analysis of several packet dumps, and the average processing time per packet is shown in Table I.

Results show that the performance obtained by NetBee and Tetheriel are very similar, respectively 75 µs and 66 µs of processing time per packet. In case only the most important information about each field are required (basically the field name, its position in the packet dump, and its size), NetBee further decreases the processing time from 75 µs/packet to 39 µs/packet. This feature is not available in Tetheriel.

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 this language.

III. STANDARD DATA EXCHANGE FORMATS FOR DECODED PACKETS

Even though NetPDL-based engines can be implemented for performing any kind of packet-based processing, a major application field is packet decoding. As a matter of fact, the first implementation of a NetPDL-based engine (available in the NetBee library) has been created for this task. Consequently, an interchange format has been specified for the output of a packet-decoding engine, which is based on PDML and PSML. In principle these languages are not related to NetPDL (besides all being based on XML); however, a NetPDL-based engine easily creates these documents because of the similarity of some elements and attributes in PDML/PSML and NetPDL visualization templates.

PDML and PSML files can be integrated with an XSL (eXtensible Stylesheet Language) file to provide a customized view of the packets. For instance, Analyzer 3.0 [1] (an open-source sniffer created by the Authors) uses a simple set of HTML/Javascript and XSL files to display a network trace into a web browser, with the same look and feel of a native, custom developed interface.

A. Packet Details Markup Language

The Packet Details Markup Language (PDML) [13] is a simple language to express information related to decoded packets (e.g. the protocols, all the field names and their values, etc.).

The Ethernet frame description in Figure 7 provides an example. A root <pdml> tag delimits the PDML document, which is a collection of packets (delimited by the <packet> tag), which includes a set of protocols (<proto> tag). Each protocol contains the list of fields (<field> tag) that have been identified in its header; the most important information for each field (name, position, size and value) are stored as attributes. A dummy protocol, geninfo, is used for information about the whole packet (ordinal position in a packet sequence, length of the packet, number of bytes actually captured, timestamp).

<table>
<thead>
<tr>
<th>Tool name</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tetheriel (native code)</td>
<td>66 µs/pkt</td>
</tr>
<tr>
<td>NetBee</td>
<td>75 µs/pkt</td>
</tr>
<tr>
<td>NetBee</td>
<td>39 µs/pkt</td>
</tr>
</tbody>
</table>

TABLE I PERFORMANCE COMPARISON BETWEEN NATIVE CODE AND NETPDL-BASED ENGINE IMPLEMENTATION
The NetBee library provides a first implementation of a set of modules for modular packet processing and it is currently used by the Analyzer 3.0 sniffer. It implements the three languages presented in this paper and it includes a first module for packet decoding, a second for field formatting (i.e. transforming an hex dump into a printable IP address and vice versa) and a third experimental module for packet filtering. This library exports a very clean interface that allows programmers to forget low-level packet processing details. For instance, Figure 10 shows the few lines of code required for packet decoding, a second for field formatting (i.e. transforming an hex dump into a printable IP address and vice versa) and a third experimental module for packet filtering. This library exports a very clean interface that allows programmers to forget low-level packet processing details. For instance, Figure 10 shows the few lines of code required for packet decoding and printing a portion of its content.

Although this library is still in the first stage, it includes a protocol database 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 library is

<table>
<thead>
<tr>
<th>Tool name</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>Packet decoding and PDML/PSML creation</td>
<td>Tethereal 1077 µs/pkt</td>
</tr>
<tr>
<td></td>
<td>NetBee 648 µs/pkt</td>
</tr>
</tbody>
</table>

IV. TOWARD MODULAR PACKET PROCESSING

The NetBee library provides a first implementation of a set of modules for modular packet processing and it is currently used by the Analyzer 3.0 sniffer. It implements the three languages presented in this paper and it includes a first module for packet decoding, a second for field formatting (i.e. transforming an hex dump into a printable IP address and vice versa) and a third experimental module for packet filtering. This library exports a very clean interface that allows programmers to forget low-level packet processing details. For instance, Figure 10 shows the few lines of code required for decoding and printing a portion of its content.

Although this library is still in the first stage, it includes a protocol database 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 library is
This library demonstrates the feasibility, the efficiency, and the simplicity (from the high-level programs perspective) of the proposed modular approach for network packet processing.

```c
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("Total length= %d\n", PDMLPacket->Length);
    printf("Protocol %s: size %d, offset %d\n",
            ProtocolItem->Position, ProtocolItem->Size, ProtocolItem->Position);
    // Read 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 10. Sample code using the NetBee library: decoding and printing the details of a packet.

V. CONCLUSIONS

This paper contains three important contributions. First, it proposes some a new form of modularity applied to packet processing. Second, it defines some new languages that can be used as standard data exchange formats between the applications that are based on packet processing. Third, it demonstrates, through the creation of the NetBee library, that the proposed approach is feasible, efficient, and it can potentially bring tremendous advantages to a large set of network applications. In fact, even though packet processing is a complex task, it has been proved extremely useful for packet processing and they have been implemented in several tools outside our research group.

Finally, the NetBee library is a modular, efficient library that provides an insight of the possibility of modular packet processing. This library has been implemented in order to demonstrate the feasibility (and the efficiency) of the proposed solutions and it supports packet decoding and an experimental form of packet filtering.

Future work will focus on improving and extending the characteristics of the NetPDL language, defining new exchange data format, and implementing new processing modules in the NetBee library. Comments and contributors are welcome.

REFERENCES