Extending the NetPDL Language to Support Traffic Classification

Extending the NetPDL Language to Support Traffic Classification

Fulvio Risso
Dip. di Automatica e Informatica, Politecnico di Torino
Corso Duca degli Abruzzi 24, Torino, Italy
fulvio.risso@polito.it

Andrea Baldini and Flavio Bonomi
Cisco Systems, Inc.
170 West Tasman Drive, San Jose 95134, CA, USA
{abaldini,flavio}@cisco.com

***Abstract*—Despite the importance of traffic classification in modern networks, the number of languages tailored to this task is extremely limited. These languages can be valuable, because they allow the update of an application (e.g. firewall) in terms of supported protocols by simply updating its protocol description database, instead of recompiling the application from scratch. This paper presents a set of extensions to the Network Protocol Description Language (NetPDL) allowing support of traffic classification from data-link to application-layer protocols. A set of preliminary experimental results obtained with these new extensions is presented as well.**

I. Introduction

The problem of traffic classification is rapidly becoming one of the hottest issues in modern networks. On one hand end-users tend to hide their traffic in order to escape Internet Service Providers or corporate rules, for example in terms of bandwidth limitation or firewall restrictions; on the other hand ISPs and enterprise network managers want to control the amount of bandwidth consumed per user and per application, limiting the most bandwidth-hungry applications or stopping some kind of traffic altogether.

Several companies proposed their solutions for packet classification. However, almost all solutions are proprietary and leave no room to user customization. This is perceived as a big problem for at least two reasons. First, a customer has to rely on an official software update to support new protocols, say a new, fast growing peer-to-peer application. Software updates can be delayed because the manufacturer has to run several regression tests to avoid any side effect; this may impact delivery time substantially. Second, the customer may have protocols developed in-house in its intranet (financial accounting is a good example) and there is little chance that off-the-shelf products can correctly classify such traffic and give it the proper rights in terms of bandwidth and security.

The NetPDL language (presented in [1]) addresses this problem by opening the path to protocol-independent applications. Figure 1 shows a possible architecture: the application relies on an external module that exports a set of functions such as `GetFieldValue(FieldName)`. When the application calls this function, the NetPDL-aware module scans the NetPDL protocol database looking for the field the application is interested in, and then extracts the offset and the size of the field. Finally, it reads the data located at the calculated offset within the packet buffer and it returns it to the caller. Once based on this model, an application can understand any protocol described in the NetPDL protocol database; it can be updated by simply improving the list of supported protocols, which is located in an external file. This model does not require the application to be thoroughly tested after the modification (in fact, the application code has not been modified at all), and offers the customer the possibility to modify the protocol database for supporting its proprietary protocols.

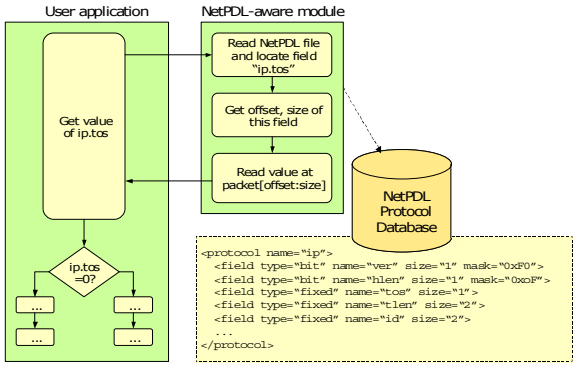

Figure 1. Possible architecture for a protocol-agnostic application.

The NetPDL language has been proved extremely effective (and very simple) for describing protocols ranging from data-link to transport layer. However, it was not very effective with application-layer protocols. This paper presents the work done to address this issue and a set of preliminary results of NetPDL-based application-layer classification.

This paper is organized as follows. Section II presents a number of solutions proposed for protocol description and their drawbacks. Section III presents the modifications to NetPDL required to support application-layer classification. Section IV presents a first experimental evaluation of the resulting language in terms of protocol coverage and performance, and finally Section V presents some conclusive remarks and a brief look at the future research directions.

II. Related Work

Cisco is one of the most active vendors from the protocol language description point of view, starting from the Cisco Flexible Packet Matching [2], included in IOS version 12.4. However, this language is rather simple and does not

inherently support traffic classification in the sense used in this paper. Perhaps the best example is the Network Based Application Recognition (NBAR) [3], which is able to define protocol formats at layers L2-L4, and, in some sense, also the format for some L7 protocols (filters such as "*match protocol http url '*root.exe**" are supported). Although its description capabilities are not perfect (it is not able to describe all the details within an application-layer protocol format), it implements some nice mechanisms that allow detecting an application by examining some packets of each connection, with some minimal state requirements. In this respect, NBAR does a deep packet inspection on packets in a session to determine which traffic category the flow belongs to. The categorization is usually done through a set of application-dependent signatures that are applied packet-by-packet. While this method is reasonably simple and it does not require excessive hardware requirements, it does not cover all possible classification cases or protocols. For instance, even when a TCP re-assembler is used within NBAR (avoiding the misdetection of a signature spanning over different packets), NBAR has a limited capability to take into account the protocol state machine, which is acceptable for a wide range of applications but may be a problem for others. Moreover, users must use binary files (Packet Description Language Module, PDLM) to program the NBAR module. The customizability of the protocols supported by the network device is limited to the addition of some regular expressions for detecting new types of traffic. In addition, this language includes both protocol description and primitives to define the action logic, e.g. "raise an alert in case of an HTTP packet". A good separation between protocol description (which is unique among different applications) and actions (which depends on the applications, e.g. packet decoding for sniffers, packet marking for QoS, and more) can give more flexibility and can make the language simpler.

A second option provided by Cisco is the Service Management Language (SML) language [4]. This language is oriented toward L7 processing and includes different parts that describe message semantics (through multiple Abstract Data Types, such as numeric, string, ASN.1 types, and more), message encoding (e.g. ASCII vs. UNICODE strings) and sequence (i.e. the protocol state machine). The user-oriented part of the language is similar to Java, and allows service providers to write their own scripts in order to customize application processing [6] (e.g. performing an action in case a given event occurs), although this task is usually done by Cisco engineers under the supervision of the customer [5]. SML mixes protocol description and actions too, with the consequent additional complexity and difficulties of supporting other applications that need different sets of actions. In addition, this language mandates the use of a TCP/IP normalizer (i.e., a module that handles IP fragments and TCP segment reordering/duplication issues and more); hence cannot be used for L7 processing on a per-packet basis.

Binpac [7], used by some dissectors distributed with BRO, is another example of a powerful language for protocol description. This declarative language supports the description of the header format, the protocol state machine, and allows correlating both directions of a flow. However, *binpac* is not very intuitive and its traffic classification capabilities are scarce (e.g. the signature-based classification must be done externally, then BinPac validates the selected protocol against its expected behavior). In addition, it mandates the use of a TCP/IP normalizer.

Another example is the L7-filter [8] project, which classifies application-layer traffic through a set of signatures. However, it does not have any protocol description capabilities (only protocol signatures) and it is not able to correlate different directions of a flow. IPP2P [9], devoted to peer-to-peer traffic classification, may be interesting as well because of its possibility to use more advanced classification criteria (e.g. signatures in addition to other mechanisms), but the classification is hard-coded in the source code and no external descriptions are used.

Among the existing set of languages, the most problematic characteristics are: difficult to understand, often proprietary, not designed to use on a per-packet basis, mostly signature-based, and often tailored to application-layer classification, without support for L2-L4 classification. NetPDL aims at being non-proprietary, with per-packet processing in mind (although it can support TCP/IP normalization), easy to understand, and yet powerful. In addition, it supports the full range of L2-L7 classification.

III. Classifying Application Protocols with NetPDL

A NetPDL database is made up of a set of protocol descriptions (element `<protocol>`), each one including a header format section (element `<format>`) and an `<encapsulation>` element, i.e. a set of expressions that return the header that follows the current one. Figure 2 shows an excerpt of the NetPDL description of an Ethernet header, which consists of 3 fixed-length fields, whose length is respectively 6/6/2 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 `<nextproto>` element) following the current Ethernet header based on the value of the `type` field.

```
<protocol name="Ethernet" longname="Ethernet 802.3">
  <format>
    <fields>
      <field type="fixed" name="dst" size="6"/>
      <field type="fixed" name="src" size="6"/>
      <field type="fixed" name="type" size="2"/>
    </fields>
  </format>
  <encapsulation>
    <switch expr="buf2int(type)">
      <case value="0x0800"> <nextproto name="#IP"/> </case>
      <case value="0x0806"> <nextproto name="#ARP"/> </case>
    </switch>
  </encapsulation>
</protocol>
```

Figure 2. NetPDL example.

Since NetPDL was originally defined for L2-L4 protocols, its protocol classification primitives were rather simple. In

fact, L2-L4 protocols usually rely on the value of some fields for choosing the next protocol, such as the *ethertype* field of the Ethernet frame that specifies if the packet is IP or ARP. Almost all these protocols can be successfully managed with some *if-then-else* elements, and most of them can even use the *switch-case* primitive, which allow defining an even simpler condition for detecting the following protocol in the packet.

Classifying application-layer protocols is much more complex, and at least three additional issues must be addressed, namely protocol verification, session tracking, and application-negotiated sessions.

A. Protocol verification

TCP/IP has an ambiguous mechanism for application-layer de-multiplexing. For instance, while a value 0x800 in the *ethertype* field uniquely identifies an IP packet, the value "80" in the TCP port does not necessary mean that the packet contains an HTTP payload, although HTTP should use the TCP port 80. A robust packet classifier should perform some form of validity check on the protocol to guarantee that the packet really is what it appears to be. Unfortunately, the verification process can be very difficult.

A first degree of verification relates to the correctness of the transmitted data from the *syntactical* point of view (e.g. a supposed HTTP payload should contain HTTP headers). A verifier should decode all the fields contained in the message and guarantee that the message is well formed, as in the case of SML and binpac. This is extremely expensive from the processing point of view; therefore regular expressions with some other protocol-dependent constraints (e.g. a payload length check for fixed-length data protocols) are often used for this purpose.

A second degree of verification relates to protocol conformance, (*protocol conformance* verification), e.g. controlling that an *HTTP GET Request* from a client is followed by a valid response from the server. This form of verification is more accurate because it can validate the run-time behavior of the protocol against its canonical state machine as defined by specifications.

A third degree of verification refers to the *semantic* of the data, e.g., the possibility to verify whether an image object transferred by the HTTP protocol is in fact an image, or some other form of content. This verification is extremely useful to detect "smart tunneling" mechanisms, in which an application uses another protocol to transport its data.

These three degrees of verification have different complexity and requires different processing capabilities. Being NetPDL packet-based and targeted to high-speed packet processing applications, only the first type of verification is well supported, although in principle it can be used also for verifying the protocol behavior.

NetPDL supports the verification through the new `<verify>` element (under `<protocol>`), which includes both an expression and a set of associated actions. The verification can either return "found" or "not found", or it can postpone the result with a "deferred" or "candidate" return code. The "deferred" is used for protocols that require several packets to be analyzed in order to return an exact answer (e.g. RTP, Skype). Vice versa, the "candidate" is used for protocols in which the payload can match several protocols at the same time. For instance, KAZAA communicates through HTTP messages that contain a special optional header; hence KAZAA packets are also valid HTTP ones. However, the NetPDL is able to differentiate among these protocols and pick the correct one (in this case, the check against the HTTP signature returns "candidate", and this protocol will be the correct one unless a check against another protocol returns "found", in which case the second protocol is chosen).

1) NetPDL and packet boundaries

One of the problems involved in protocol verification is that, at application level, the payload may span over several packets. Even if NetPDL is packet-based, it is agnostic with respect to the definition of packet. For instance, a packet can be either a frame (as transmitted on the Ethernet) or a *virtual packet*, i.e. a buffer that contains all the payloads put together by a TCP normalizer, as shown in Figure 3. In case an Ethernet frame is processed, the NetPDL engine cannot execute any operation beyond the packet payload; vice versa, if the TCP normalizer puts together all the pieces of a message and delivers the resulting buffer to the NetPDL engine, this will be able to perform its verification controls over the appropriate amount of data. However, NetPDL does not have any primitives for TCP normalization; hence some external module (from NetPDL point of view) must address this task in case we want NetPDL to operate on messages instead of packets..

The problem of data reassembly can be present at several layers, e.g., several ATM cells that contain an IP packet, IP fragmented packets, applications whose payload spans over several TCP packets; in any case, this is outside the scope of the NetPDL language. The user must take care of pre-formatting the data in such a way that the NetPDL engine gets a buffer with the entire set of data it expects for that protocol.

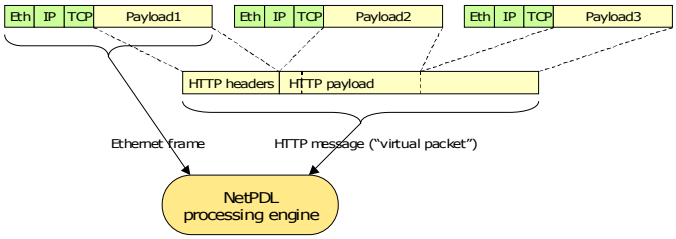

Figure 3. Processing *real* and virtual packets in NetPDL.

2) Verification vs. classification

Protocol classification is usually done only through the syntactical analysis. For instance, the packet payload (or a reassembled payload, in case of large messages) is often compared against a set of signatures, which characterize the protocol that generated that message. Signatures are used because of their speed, since complete syntactical verification may be extremely slow.

Once the protocol has been identified, there is nothing more to do from the classification standpoint; a more sophisticated verification process is targeted mostly at checking that the

protocol has a correct behavior, which is useful to protect the network from crafted messages that exploit security vulnerabilities in hosts.

Although this cannot be taken as a "hard" rule, we can say that the accuracy of the verification depends on needs: the syntactical verification is enough for classification purposes, while the protocol conformance verification is useful mostly for security purposes. Conversely, the semantic verification can be very useful for classification (e.g. to protect from applications implementing smart tunneling techniques), but at the best of the authors' knowledge, there are no technologies that are able to address this issue at this time.

B. Session Tracking

Session Tracking is mostly used to keep track of TCP sessions. This mechanism leverages a simple table containing the 5-tuple that includes the ID of known sessions and the associated application-layer protocol.

The use of the session tracking is often different in packet-based technologies (e.g. NBAR) and in stream-based technologies (e.g. SML, BinPac). In packet-based technologies, the session table stores the result of the signature matching; for instance, signatures are usually present only at the beginning of the session, so the result must be saved to associate following packets of the same session to the protocol detected in the previous step. When a packet belongs to a known session, the protocol signature should not be verified again and the selected protocol should be used for further processing. By contrast, in stream-based technologies the session table is used to retrieve a pointer to the given session, perform the TCP reassembly and jump to the correct L7 protocol analyzer.

In order to implement the session tracking, NetPDL defines a special bi-dimensional variable (element `<lookuptable>`) that supports an arbitrary number of fields. Fields are either keys to locate entries ("primary key" in database terminology) or data (such as protocol ID) related to the given element.

Although bi-dimensional variables can have any use, they are particularly useful for transport-layer session tracking. These entries (e.g. TCP sessions) have the necessity of being properly managed, e.g., we must be able to purge "zombie" TCP sessions that are no longer active. For this reason, NetPDL can associate an attribute to each entry, defining its validity. An entry can last forever (unless deleted by an explicit command in the NetPDL file), or it can be automatically cleared off after a given inactivity time.

C. Application-negotiated sessions

The third problem is the case of applications that dynamically negotiate the parameters of the session, e.g., the case of FTP data connection whose ports are dynamically negotiated in the FTP control channel, or SIP sessions that dynamically negotiate RTP ports.

NetPDL supports a set of processing elements through a new `<execute-code>` section (under `<protocol>`). For instance, the definition of the FTP protocol will contain a piece of code that recognizes the negotiation of a new FTP data session, and inserts a new entry into the TCP session table. Usually these entries do not have to go through a verification process – i.e., if the "master" session is trusted (it has already been verified before), its "child" sessions should be trusted as well.

One additional problem related to this point is that often the entire 5-tuple is not known in advance. For example, the PASV command used in FTP passive connections leaves one of the TCP ports unknown. NetPDL supports also the insertion of partial entries – i.e., entries in which part of the primary key is missing; the entry can be automatically replaced with a complete one as soon as a session matches. This behavior is highly customizable and there are cases in which the partial entry can still stay in the session table. For example, a protocol processing logic may create the TCP session table in which an entry such "ip_unknown, port_unknown, ip_cisco_com, port 80 → protocol HTTP" is statically allocated at time zero. This rule allow immediately to associate any TCP packet directed to the cisco.com IP address on port 80 as belonging to the HTTP protocol, and it should not be replaced when an hit occurs.

The capabilities in terms of tables provided by the NetPDL language are rather sophisticated; for more details, please refer to the NetPDL documentation [10].

IV. Experimental evaluation

The NetPDL language went through a major revision compared to previous versions [1]. In addition to the elements that were briefly introduced in the previous section (for more details, please refer to the NetPDL documentation [10]), there was a major rewrite of the syntax related to protocol fields and expressions (it uses a more readable syntax, no longer based on XML, with typed operands), and a better cleanup of the main NetPDL sections (e.g., the new section devoted to code processing, the `<execute-code>`). Additional improvements include the possibility to define external processing handlers (when a given element of the NetPDL file is encountered, the processing can continue to a function defined into an external program), and a major rewrite of the API used by the Packet Decoder module within the NetBee library [11], which is the most advanced tool based on NetPDL nowadays.

A. The TCP/UDP encapsulation section

After the language basics, presented in Section III, this section presents an example on how the new elements can be combined together to create a piece of code that performs packet classification. This paragraph presents perhaps the most significant example of protocol classification, namely the skeleton of the `<encapsulation>` section of the TCP protocol (shown in Figure 4), which uses the new elements but also takes care of performance issues, as explained later.

When a TCP packet is found, the NetPDL code extracts the session ID and performs a lookup in the TCP session table. If the session is already known (what we call *dynamic entry*), the processing continues with the protocol stored in that record (element `<nextproto>`). In the opposite case, the NetPDL executes a section that contains what we call *well-known entries*. This section verifies (`<nextproto-candidate>`

element) if the packet contains a protocol usually associated to a well-known port (e.g. HTTP in case of port 80). If the protocol is still unknown, the NetPDL lists a set of protocols that must be checked against the payload (what we call *"try and see" entries*), until a match is found. In the unfortunate case that the protocol is still unknown, the packet is associated to a default protocol.

In theory, "well-known" entries are not necessary. In this case the "try and see" section will be executed for all the new sessions, but this results in a performance deterioration since multiple signatures have to be checked till the correct one if found. Vice versa, the "well-known entries" section allows testing the most likely signature first; only if this step fails the execution continues with the "try and see" section.

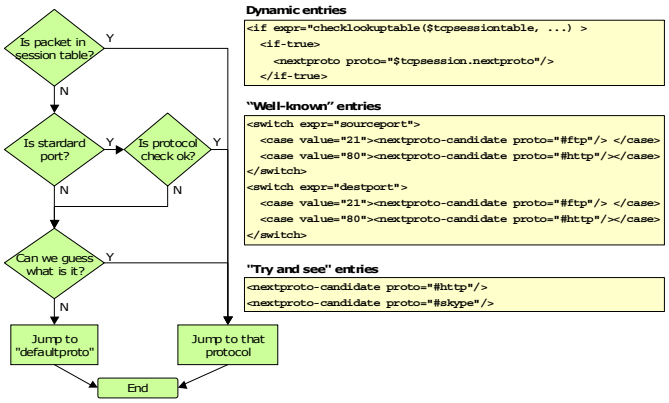

Figure 4. Structure of the packet classification section of the TCP protocol.

B. *Processing complexity*

Although a more detailed evaluation of the performance of NetPDL (in terms of accuracy of protocol classification and protocol coverage) is deferred to a following paper, Table 1 presents some performance results in terms of processing cost per packet. It refers to a trace of about 9GB containing 15M packets, captured at the exit link of our University. The file was first processed deleting all the packets that belong to sessions that started before the beginning of the capture. Tests were performed on a standard Pentium 4 PC clocked at 3.0 GHz with 2 GB of memory and made use of the `PacketDecoder` sample provided in the NetBee library, which decodes packets associating each field with its value and its position in the packet dump (this excludes all the processing related to NetPDL "visualization primitives").

The first result refers to the processing cost with the previous version of the NetPDL language, in which packet classification was extremely simple (port-based). The second relates to a NetPDL file in which the session tracking has been turned on, and the result is significantly better because of the faster classification for packets belonging to known TCP/UDP sessions (a lookup in the session table instead of processing the entire encapsulation section). In the third case, the session tracking is coupled with protocol verification capabilities, and the result shows that the protocol verification makes the processing cost almost three times higher, due to the extensive usage of regular expression. The fourth case enables all the features (including the "try and see" section), and the cost is even higher because of the need to compare the payload against a multiple set of signatures in order to find out the correct protocol. Note that this fourth cost is still lower than the original first result, obtained forcing the classification to be done on all the packets.

Although the number reported in the first case is higher than the one shown in previous experiments [12] (mostly due to a larger set of supported protocols and a bigger capture file that does not fit in cache), the results demonstrate that the additional capabilities of the NetPDL language do not dramatically increase the processing cost. Interestingly, the session tracking capability is even able to decrease the processing cost due to fast protocol classification for packets belonging to a known session.

TABLE 1
NETPDL PERFORMANCE EVALUATION

Test mode	Processing cost
Old Netpdl, no session tracking	265 µs/packet
Session tracking, no protocol verification	41 µs/packet
Session tracking with protocol verification	110 µs/packet
Session tracking with protocol verification and "try and see" entries	243 µs/packet

C. *Protocol coverage*

At the time of writing, 122 protocols are defined using the current NetPDL specification. This number can definitely increase, since it depends mostly on the effort required to understand a new protocol and write the corresponding description in NetPDL. From the protocol coverage standpoint, NetPDL should be able to classify all the protocols that are handled by similar technology, such as l7filter and NBAR, which are mostly based on signatures. Currently we support many IETF protocols, and several VoIP and P2P applications. With the current set of protocols and the current signatures, the classification process performed on the previous capture trace returns the result shown in Table 2, in which most packets are associated to a L7 protocols and only about 3% of them are unknown.

At this time, the authors do not have any number to demonstrate the accuracy of the classification, which is left to future work; however, being the classification methods similar to previously cited technologies, we do expect to have accuracy equal or better than them. For instance, L7-filter does not support RTP (Real Time Protocol) because it cannot be detected through signatures, while NetPDL has full support for it. In addition, NetPDL supports several protocols that negotiate other connections at run-time, such as SIP, FTP, and more.

TABLE 2
NETPDL CLASSIFICATION EXAMPLE

Protocol	Number of packets	Share (%)
Edonkey over TCP	5803125	35.31%
Samba	3847739	23.41%
HTTP	2352677	14.31%
Microsoft SQL Server	1335405	8.12%

Edonkey over UDP	826500	5.03%
TCP (e.g., 3-way handshake)	634865	3.86%
Unknown	552486	3.36%
Other recognized protocols	1083580	6.59%

D. Readability

The readability of NetPDL files plays an important role in the language definition. Since creating a GUI that helps to define these files is a difficult process, the easiest way of updating the definition is to edit files by hand.

Obviously, the increased number of XML elements and attributes present in the language and the complexity of the classification make the readability worse than in previous versions. However, we feel that the general readability is still good; the `<format>` section (which contains the list of the fields of the protocol) has only been slightly modified, and the `<encapsulation>` section remains the same for most protocols, with the notably exceptions of TCP and UDP. Also in these cases the readability is still excellent. Unfortunately, some more elements are required for session management and for protocol verification; since these look more like programming instructions, they have been located in the new `<execute-code>` section. Although the processing code is not something easier to read when formatted through XML element, the readability of most of these sections still looks acceptable.

Although in principle the `<execute-code>` section can accommodate any type of code (since the processing primitives of NetPDL are now similar to the ones in any high-level language such as C), the authors believe that the code in that section should be kept at the minimum, and more advanced processing code (e.g., statistical analysis for improving the classification) should be written elsewhere with a more appropriate language. This is now possible through external callbacks, which pass control to an external program when a NetPDL element with the *callhandle* attribute is found.

E. Implementation-dependant limitations

The current engine implementation does not have TCP/IP normalization capabilities; hence the verification process is limited to a single packet. This leads to some false negatives related to the syntactical validation (a protocol signature split across two packets makes this protocol undetectable), but still allows to perform protocol conformance verification (the engine can correctly classify sessions using several packets satisfying a set of conditions), which in fact is implemented in some dissectors, e.g., RTP, Skype.

One point deferred to future work is the evaluation of classification accuracy, also in case a TCP normalizer is used. However, as previously said, the TCP normalizer is out of scope of the NetPDL language and it is a matter of engine implementation.

V. Conclusions

This paper presents a set of extensions to the NetPDL language allowing the implementation of application-layer classification. These new language primitives are simple (albeit powerful) and lead to protocol definition files that are still quite readable.

Performances obtainable with these new extensions are interesting: the protocol coverage is pretty high, and a first implementation (far from being optimized) is already processing packets at a reasonably speed.

Future work on this topic will focus on a deeper evaluation of the accuracy of the classification achievable with NetPDL, with and without TCP/IP normalizer. Another point will be a more accurate examination of the descriptive capabilities of the language related to application-layer protocols, to understand if the format description capabilities defined for L2-L4 protocols are still suitable for L7 protocols.

Acknowledgment

The authors wish to thank Mario Baldi, Satish Gannu, Pere Monclus, Olivier Morandi, and Bob Olsen, for their comments and their many suggestions, and Christian Novello, who spent part of its graduation thesis working on these issues.

References

[1] Mario Baldi, Fulvio Risso, "NetPDL: An Extensible XML-Based Language for Packet Header Description", *Elsevier Computer Networks Journal (COMNET)*, Volume 50, Issue 5, Pages 688-706, April 2006.

[2] Cisco Systems, Cisco IOS Flexible Packet Matching; included in Cisco IOS 12.4. Available at http://www.cisco.com/en/US/products/ps6723/products_ios_protocol_group_home.html.

[3] Cisco Systems, "Network Based Application Recognition" (NBAR). Available at http://www.cisco.com/en/US/products/ps6616/products_ios_protocol_group_home.html.

[4] Opher Reviv, "Inside network programming with SML", *EE Times*, Aug 2003. Available at http://www.eetimes.com/story/OEG20030818S0077.

[5] Alex Goldman, "Control P2P Traffic", ISP-Planet, April 2003. Available at http://www.isp-planet.com/equipment/2003/p-cube_engage.html.

[6] Cisco Systems, "Service Control Application Suite for Broadband", API Programmer's Guide Ver. 2.5.5. Available at http://cco.cisco.com/application/pdf/en/us/guest/products/ps6135/c1671/ccmigration_09186a00804249 11.pdf.

[7] Ruoming Pang, Vern Paxson, Robin Sommer, Larry Peterson, "binpac: a yacc for writing application protocol parsers", Proceedings of the 6th ACM SIGCOMM on Internet Measurement, Pages: 289-300, Rio de Janeiro, Brazil, October 2006.

[8] L7filter, Application Layer Packet Classifier for Linux, May 2003. Available at http://l7-filter.sourceforge.net/.

[9] The IPP2P project. Available online at http://www.ipp2p.org.

[10] Fulvio Risso, "NetPDL Language Specification". February 2007. Available at http//test.nbee.org:8080/netpdl/.

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

[12] Mario Baldi, Fulvio Risso, "Using XML for Efficient and Modular Packet Processing", *Proceedings of IEEE Globecom 2005*, St. Louis, Missouri, USA, December 2005.