

**Programmable In Network Capability
for Streaming Device or Cloud Gateway**

David Bernstein
Cisco Systems, Inc.
daberns@cisco.com

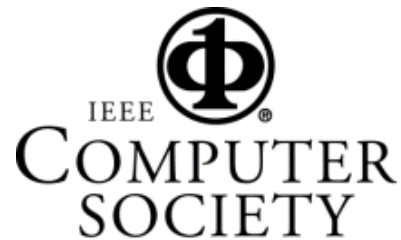
John McDowall
Cisco Systems, Inc.
jmcdowal@cisco.com

Krishna Sankar
Cisco Systems, Inc.
ksankar@cisco.com

Stanley Poon
Cisco Systems, Inc.
spoon@cisco.com

As Published in

SERA 2008
6th International Conference on
**Software Engineering Research, Management and
Applications**
August 20-22, 2008
Prague, Czech Republic



Programmable In Network Capability for Streaming Device or Cloud Gateway

David Bernstein
Cisco Systems, Inc.
daberns@cisco.com

John McDowall
Cisco Systems, Inc.
jmcdowal@cisco.com

Krishna Sankar
Cisco Systems, Inc.
ksankar@cisco.com

Stanley Poon
Cisco Systems, Inc.
spoon@cisco.com

Abstract

Many network devices implement capabilities to manipulate traffic depending on the application. Examples include a firewall or a load balancer. These are based on Layer 2-4 packet-based classifications such as port or protocol, or signature recognition. Although configurable or extensible via scripting, they are not generally programmable. We present an architecture extending classification to programmable, semantic Layer 5-7 capabilities. The architecture has programmable handling of that classified traffic, which are message flows, not packets. This has led us to a new, in-network message streaming based programming model. Finally, we present a series of network platform capabilities delivered as components, from precision timing to programmable QoS to network identity. Use cases considered range from traffic shaping to Cloud Gateway.

1. Introduction

From the early days, TCP/IP networks have relied on deep packet inspection techniques [1] [2] to implement key, in-network functions. Deep packet inspection along with layer 4 TCP proxy capabilities were the key technologies enabling the firewall. By being able to sit in-line in the network transparently, the firewall could understand “6-tuple” parameters and implement configured policies (usually *allow* or *block*). 5-tuples consist of 1) source address, 2) destination address 3) source port number 4) destination port number 5) protocol, and more recently 6) VLAN.

Much work has been done on accelerating this processor intensive task [3], in extending the classification capabilities beyond the 5-tuple to include application “signature” awareness [4] [5]. This work has enabled innovative network devices including firewalls, intrusion detection, and load balancers to emerge.

These types of network devices are configured by network administrators to incorporate policies for actions once the traffic flows are classified. The mechanisms by which the policies are expressed are closely related to

the fixed-function capabilities which the devices are designed to perform. Load balancers are designed to redirect traffic, most frequently web browsing traffic, amongst a set of web servers using an administrator specified policy.

They allow customization and extensions of the policies by providing interfaces to external systems, for example allowing high-priority subscribers to be routed to one set of less-busy servers whilst low-priority subscribers can be routed to the more-busy servers. Additionally custom business logic can be accommodated to some extent through scripting capabilities [6].

2.1 Traffic classification challenges

Traffic on networks has both exploded and evolved to adapt to the limitations and configuration of corporate LANs and of the public internet. For example, consider the impact of the proliferation of firewalls. They are most commonly configured to block all but web traffic, which means that if an application protocol uses anything other than HTTP/HTTPS transiting over ports 80/443, the likelihood of getting through is quite low. This has caused most applications to adopt a tunneling or encapsulation strategy putting the application protocol inside of HTTP.

As deep message inspection along with application signature technology improves, so do the applications. For example, to obscure the payloads inside of HTTP applications may adopt multiple levels of encapsulation, using binary coding for their payloads. Some traffic, which service providers and enterprises wish to block, have become even cleverer. In the case of Peer to Peer applications (file sharing, video sharing) several obfuscation techniques, including dynamic port numbers, port hopping, HTTP masquerading, chunked file transfers, and encrypted payloads are all utilized [7].

Simple signature techniques can be fooled because a single additional encapsulation, a new binary encoding algorithm, or other simple techniques will yield a completely different signature.

One can only classify this type of traffic by manually decomposing it, and creating a semantic/behavioral

definition file for it. In this way, just as the recipient of the traffic understands it so can the intermediary. This leads one to a need for L4-L7 deep inspection. Inspection at these upper layers is called “deep *message* inspection” as L4-L7 calls for the TCP stack to have reconstructed the packets into message flows.

2.2 Programmability challenges

Even with all the advances in traffic classification, there has been little advancement in device programmability. Networking vendors, who are the ones building this equipment, have a business model which is tuned towards fixed functionality, purpose targeted devices; programmable general purpose “network platforms” are virtually non-existent.

This is tied directly to the “classification challenge” which is outlined above. It is one thing for a networking company to create a purpose built set of functions which operate on packet traffic. However applications developers are interested in message flows, which would provide them the same sort of reconstructed traffic they would find sitting on top of a TCP/IP socket. It is precisely the complication and processing overhead of implementing deep *message* inspection which makes the exposure of an applications programming interface heretofore absent.

Recently, several networking vendors have announced open interface interfaces to their embedded operating systems [8] [9] [10] [11] but these are not generalized programming environments supporting compiled languages, and are specific to the vendor platform.

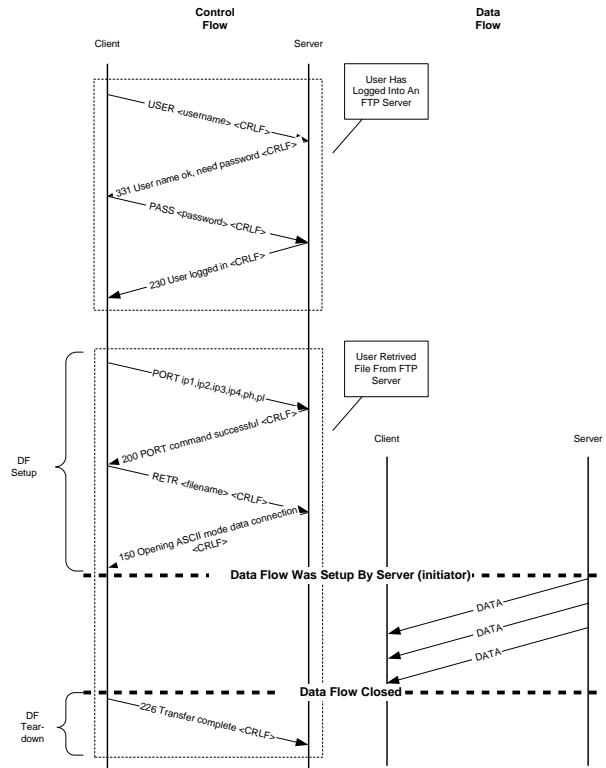
In addition to the need for a message flow as a key element of a programmability model, the notion of control is important. For example, application developers are not interested in simply developing instrumentation, where message streams are observed but there is no action which can be taken. Just as the core operation of a firewall, load balancer, or intrusion detection system requires a point of action at the point of detection, an application developer would need that capability for their feature. Putting applications code in-line in the network is something networking vendors have not explored to date.

3. Message flow classification

We begin by presenting a solution to the deep message inspection challenge. At the core of this is a definition language with which one can specify message flows of interest.

3.1 Classification using a semantic message definition language

To solve this problem, we use a programming language we call SML which describes the interactions of higher level applications in terms of observing the actual on the wire traffic. To provide some introductory insight into this mechanism, consider the FTP



transaction diagrammed in Figure 1.

Figure 1. File transfer transaction

One may note that while the FTP session owns a single control flow (CF) to begin with, the process of transferring a file involves the creation of a transient data flow (DF) whose 5-tuple is negotiated dynamically over the CF (using the FTP PORT message). No prior knowledge can be utilized to pre-determine the 5-tuple of the transient DF. Thus, an application transaction is not limited to a single flow. It may employ multiple flows whose 5-tuple is determined dynamically through a message-based negotiation between the two end-points.

Following this, here is a code fragment which is going to monitor FTP traffic looking for a partial set of commands/replies pairs. Each pair generates a report data record (RDR) containing the command name, reply code and the reply’s additional text.

```

import com.cisco.lib.protocols.IETF.FTP;
const int8 LISTNER = 0;
const int8 FTP_COMMAND_NAME_MESSAGE_CODE = 1010;
const int8 FTP_REPLY_MESSAGE_CODE = 1020;
const string cmdNames[18] = {
    "USER", "PASS", "ACCT", "CWD",
    "SMNT", "RETR", "STOR", "APPE",
    "RNFR", "RNTO", "DELE", "RMD",
    "MKD", "LIST", "NLST", "SITE",
    "STAT", "HELP" };
event [publish = yes] CommandReplyPair ()
{
    template FTP_COMMAND_GROUP_STR command;
    template FTP_REPLY reply;
    while(true)
    {
        match [var=command, dir=>]()
        {
            report(DEST_RDR,
                FTP_COMMAND_NAME_MESSAGE_CODE,
                "FTP command is: ",
                cmdNames[command.cmd._curChoice],
                " with single parameter: ",
                command.arg);
            match [var=reply, dir=<]()
            {
                report(DEST_RDR,
                    FTP_REPLY_MESSAGE_CODE,
                    "Reply code is: ",
                    reply.code,
                    " with reply text: ",
                    reply.text);
            }
        }
    }
    success();
}

```

Looking for the command message, the match statement may either succeed, or fail looping back to the top to wait for the next message. However once found, execution continues into the match statement success clause, invoking the report action. The report, in turn, sends out an RDR (of type FTP_COMMAND_NAME_MESSAGE_CODE) containing, among other things, the name of the FTP command, and the command's argument.

Going deeper, we have continued by defining the rest of the areas of interest in FTP, for example, data transfers, etc. In this way one can look at who is doing the FTP, where from and to, and what actual content. It is this kind of processing capability which is required to understand application traffic in general.

3.2 Classification using regular expressions

When one tries to write classification expressions covering text formats, a regular expression like capability is needed. SML implements reserved words directly as "Abstract Data Types" (ADT) in the language. ADTs are a direct and convenient way to do

regular expression operations. ADTs come in different types, one being string based i.e. regex another being Abstract Syntax Notation One (ASN.1) [12]. ASN.1 is a notation for regular expression-like specification which has special operators that ledn themselves to regular expression like parsing. The subset of ASN.1 supported in SML ADTs are shown in Table 1:

BOOLEAN,	ALL	PRESENT
INTEGER	APPLICATION	TRUE
BIT	PRIVATE	FALSE
OCTET	BMPString	PLUS-INFINITY
NULL	GeneralString	MINUS-INFINITY
SEQUENCE	GraphicString	CHARACTER
OPTIONAL	IA5String	STRING
DEFAULT	ISO646String	REAL
COMPONENT	NumericString	UTF8
OF	PrintableString	COMPONENTS
SET	TeletexString	IMPLIED
CHOICE	T61String	OBJECT
IMPLICIT	UniversalString	IDENTIFIER
EXPLICIT	VideotexString	DEFINITIONS
ENUMERATED	VisibleString	BEGIN
UNIVERSAL	MAX	END
INCLUDES	SIZE	AUTOMATIC
MIN	FROM	TAGS
INTERSECTION	WITH	IMPORTS
UNION	ABSENT	EXPORTS
EXCEPT		EXTENSIBILITY

Table 1. ASN.1 ADTs

3.1 Classification using XML Path Language

Much content today are represented as XML [13]. In particular, XML is used extensively inside of Web Services [14]. As such, if one wants to understand this traffic one must look into an XML document using a well know XML-specific content specification. XML Path Language [15], or XPath, is a language for addressing parts of an XML document.

In an in-network environment, one can not use the typical Document Object Model (DOM) [16] processing techniques, as these are server model, in that they assume that the entire document is in memory and one can search forward and backward in the XML DOM. In a message flow which is a TCP proxy, of course, one wants to process in a stream mode, as fast as possible in order not to provide back pressure network congestion.

We implemented a streaming XPath model. The basis of this was YFilter [17] [18] [19] which has been adopted in several network implementations and is relatively simple to implement with good performance characteristics. It is scalable to large set of XPath and can be enhanced to handle predicates for actions.

Our system, which we called Zfilter, is based on an XPath compiler approach, where we parse individual XPath expressions, validating that the expression conforms to a streaming subset. We generate an NFA for the set of expressions, and then optimize the NFA for efficient runtime execution.

The interface to the Zfilter engine is listed below:

```
void zf_init();
error_code_t zf_compile_expr (const zf_char_t
**expr, int length, const zf_char_t
**namespaces, int namespaces_length, zf_nfa_t
**nfa)
void zf_free_nfa (zf_nfa_t *nfa)
error_code_t zf_create_eval_context (zf_nfa_t
*nfa, zf_eval_context_t **ctx)
error_code_t zf_parse_chunk (zf_eval_context_t
*ctx, const char *chunk, int size, int
terminate, int *results_matched)
error_code_t zf_get_query_results
(zf_eval_context_t *ctx, int query_id,
zf_query_results_t **results)
void zf_free_query_results (zf_eval_context_t
*ctx, zf_query_results_t *query_results)
void zf_free_eval_context (zf_eval_context_t *
ctx)
```

4. Message stream programming model

Up to this point we have been describing classification. There is an intermediate step where the network service is loaded into a software runtime in the device, and traffic which needs to be classified is redirected through that runtime. The specifics of the runtime, and the lifecycle of the loaded service will be covered in a later section of this paper and are largely an implementation detail. To continue to fully understand the work, we turn to the programming model itself.

4.1 From packets to messages

Core to the programming model is the concept that the service is in-line in the TCP flow. This has several implications. The runtime needs to accomplish classification efficiently, so there is initialization and set up for a service, where the service informs the runtime of the traffic it is interested in. An element of the runtime called the “Flow Handler” spreads the classification into successively narrower and more specific tasks. For example, first the L2-L4 classification is done, then Crypto processing is handled (if needed, covered in later section), then the protocols themselves are decoded, asking for additional packets as necessary to form message flows, then the semantic/behavioral classification is performed, and finally for the content itself (as needed) any regular expression or XPath processing is done.

At the end of this process, a message stream is delivered to the service. It is important to understand that an application programmer who is used to developing client or server code, always has had the benefit (from a TCP/IP stack and socket interface) of seeing messages. Thus the programming model must not present a packets interface.

4.2 Programming Model

It was clear that the more one could leverage existing patterns and practices in the applications software programming space the easier it would be for developers to adopt this new capability. The model we chose is based on Java Standard Edition, with minimal set of new APIs. Our goals were to provide easy access to selected inline network traffic and to make the common cases easy by simplifying and streamlining typical flow control actions. We would provide stateless event handlers and use zero copy packet interfaces.

The most logical model was the Java “servlet” model where the servlet was invoked by a servlet container when HTTP traffic meant for the servlet arrived. We invented a “netlet” model where the netlet was invoked when the classified message stream which the netlet asked for arrived.

We similarly adopted a container model which holds and manages the lifecycle for the netlets. As would be expected in a Java environment, deployment descriptors bind netlets to the classification stack. And because there will be literally millions of flows, the model supports multiple netlets and multiple containers. Of course the containers include a JVM. A single netlet can handle thousands of flows, we need multiple netlets for different classification events and multiple containers to allow different users to run in the same device; again similar to the servlet model. The core netlet programming model is expressed below:

```
void onClassification (Message msg, Action
action)
void onMoreData (Message msg, Action action)
void onFlowEvent (FlowEvent evt, Action action)
void init (NetletConfig config)
void destroy ()
```

The first interface is the primary event handler upon message classification. Actions are discussed below. The next interface is a way for the netlet to ask for more data to enable the application developer to walk deeper into the message. The next interface is a notification about an out-of-band change to a flow state, for example, a network cable has been disconnected. Note, netlets cannot stall the network as they are asynchronous. If they take too long they will be timed out by the network. The OnFlowEvent is to let the netlet know that the network flow has been reset or timed out and it should cleanup. Finally, the interfaces for netlet registration, setup, and clean up are there.

4.3 Actions

Although the netlet service processes and makes decisions based on message flows, actions must be implemented on packet streams. For example if a flow is to be redirected this is actually effected in the 5-tuple and implemented by L2-L4 processing engine. This engine will act in an in-line mode, passing the traffic along (or not, or as modified). Or the engine can act in a monitoring mode, where actions act as a dynamic filter to continually filter the traffic the service is interested in, in real-time. These actions are configured by the netlet itself. Table 2 lists the possible actions and attributes (M=Monitoring case, I=In-line case):

Action	Description	Action Point	M	I
Drop-Silent-Current	For the current flow silently discard packets until the source closes the connection.	Post Classify	N	Y
Drop-Silent-Future	On future flows that match set of L2-4 attributes silently discard packets until source closes connection. Don't send packets for these flows to the netlet.	Pre-Classify	N	Y
Drop-Reset-Current	For the current flow send a TCP Reset to the source and close the connection.	Post Classify	N	Y
Drop-Reset-Future	On future flows that match set of L2-4 attributes send TCP Reset to source and close connection. Don't send packets for these flows to the netlet.	Pre-Classify	N	Y
Re-Direct-Current	For the current flow re-direct it to a new IP address/Port.	Post-Classify	N	Y
Re-Direct-Future	For future flows that match set of L2-4 attributes redirect them to a new IP address/Port.	Pre-Classify	N	Y
Bypass-Current	Take current flow being processed and forward original destination unmodified. Don't send packets to netlet. In case of monitoring discard future packets; for inline forward to original destination.	Post-Classify	Y	Y
Bypass-Future	For future flows that match set of L2-4 attributes don't send to the netlet. In case of monitoring discard them; for in-line forward to egress with no operations.	Pre-Classify	Y	Y
Re-write	The flow specified has been rewritten and therefore the packets being sent are new and need to be formatted for TCP or UDP.	netlet	N	Y
Time-stamp	Place timestamp in the packet header by inserting a channel that will get a system level timestamp. Control can be modified by specifying an ACL mask and port number that will be applied to all future flows.	Pre-Classify	Y	Y
Release	This is when the TCP-Proxy is instructed to release the flow but still send packets to the application. Here the service is performing traffic shaping or accounting and needs to view packets/messages even after the flow has been released.	Post-Classify	N	Y
Restore	When a flow has been dynamically modified by the netlet this action will restore it to its default state.	Pre-Classify	Y	Y

Table 2. Flow actions

5. Network services

There are support libraries and services – the “components” - surrounding the netlet runtime environment that resides in the container.

5.1 Traffic encryption/decryption services

If the traffic is encrypted, in order for classification to occur, it must be decrypted. This is the case for SSL/HTTPS traffic. To support this, an implementation platform will include a decryption processor (hardware), and the software engine will handle key management as a “man in the middle” so that on egress, the traffic is re-encrypted, and the destination sees no change.

5.2 IEEE 1588 precision timing

Many applications such as Intrusion Detection Systems, Anti Money Laundering, and Complex Event Processing, require a precision time stamp. Synchronization of network devices and millisecond time accuracy spread over a large network, with multiple points of measurement and the statistics of combines drift, make traditional, NTP [20] insufficient.

We have participated in the recent advances in IEEE 1588 standard [21] network time synchronization and approaches to interface to it [22] [23]. The system we have developed includes such capabilities allowing for precision time stamping.

5.3 IEEE 802.1x network identity

An attribute of applications traffic is “on whose behalf” it is. If an API can be made available to a service where it can be discerned who the person is on behalf of which this message flow is transiting, then the service can take particular action based on that information. This project involves experimental API's which encompass network identity derived from the IEEE 802.1x family of protocols [24] [25]. Network identity is exposed to the netlet services which can take special action based on that information. For example, this allows Network Identity to be correlated with application identity to provide the possibility of new approaches to single sign-on using the network.

5.4 RFC 2547/2702/3031 (MPLS) QOS traffic engineering

Many networks are “traffic engineered” using Multi-Protocol Labeling System (MPLS) [26] [27] [28]. These are tags carried along in the packets, and the tags are

recognized by the switches and routers in the network to provide different QoS treatment to the traffic depending on the tags.

It is quite difficult to configure and manage an MPLS network due to the fact that multiple devices which are geographically distributed need to be configured in such a way that the various simultaneous traffic flows can be differentiated in QoS but in fact many times are traveling over common links. It is interesting to applications developers to be able to change the traffic labels on the fly to have traffic transit different MPLS routes dynamically depending on the traffic (actually *message*) content.

This project involves experimental API's which is exposed to netlet services which can change MPLS tags depending on classification or content found in the messages.

5.5 RFC 3917/3955/5101/5102 (IPFIX) IP Flow Information Export

When a netlet service wants to communicate with the outside world in terms of reporting timestamps, traffic statistics, actions taken, or any other information, interfaces to a standard facility called IP Flow Information Export (IPFIX) [29] [30] [31] [32] is implemented.

There are many IPFX compatible libraries and utilities which can be used with this system as a result.

6. Logical Architecture

Figure 2 illustrates the logical architecture:

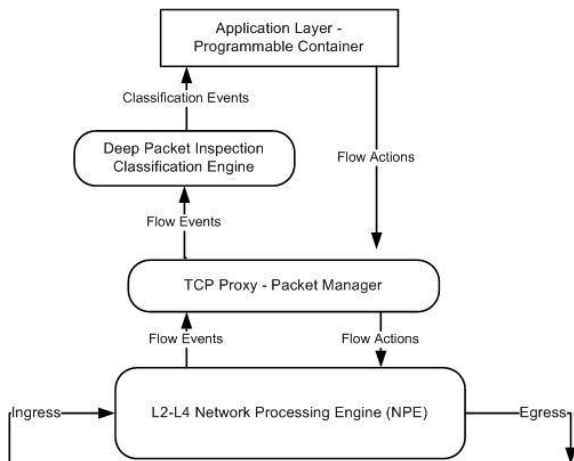


Figure 2. Layered logical architecture

As can be seen, layered architectural approach was used, with the Network Processing Engine (NPE) as the bottom layer, interfacing via a Packet Manager abstraction to the Deep Packet Inspection Layer which in turn provides the programming model (4.2) to the applications layer.

The layering approach also facilitates the separation of capabilities. The NPE does the packet processing like 5-tuple classification, packet re-writing, packet redirection, handling network protocol state machines et al. The Deep Packet Inspection and Classification Engine deliver implements the classification, reconstruction into message streams, ADTs etc. Finally the application service layer implements the application logic.

6. Component lifecycle

In order to develop, test, and then deploy the system, several facilities had to be developed.

Development utilizes standard Java IDEs with container class library interface definitions so that netlets can be developed using standard tools and on a desktop. There are helpers to load netlets into containers.

Testing requires that a network stream replay be enabled so that netlets can be provided preconfigured traffic to see if they and the classification machinery around them functions as expected.

Deployment facilities to load, start, stop, and remove containers and netlets and to instrument their running behavior based on standard JMX [33].

7. Performance

An essential aspect of such a system is the performance, especially how we bridge the impedance mismatch between the wire-speed and the application processing.

Single copy and multi-core optimization techniques are used everywhere. The layered architecture enables us to process the packets at the network layer – thus keeping a single copy – while providing pointer reference to the upper layers. Moreover the flow model enables us to leverage multi-core capability natively.

The architecture leverages hardware acceleration as and when available and needed. If there is hardware present it will be fast crypto processors, 5-tuple classification capability in the Network Interface Cards, or ASIC based classification engines for regular Expressions for example. We can seamlessly and transparently leverage the various hardware capabilities available in the run-time environment by presenting a consistent programming interface, hiding the software or hardware implementation.

The architecture separates throughput and latency effectively by moving back and forth between the concepts of packets, messages and flows. The latency is introduced for first few packets as the applications layer calculates the semantic context of a message. Once classified, the application layer instructs the NPE to handle the flow thus achieving full throughput.

A stream processing technique is used. Any other model will impact the performance and add complexities only a server can handle; such capabilities are best hosted in a server.

8. Conclusions

This work was driven by a desire to allow developers to put a part of their application in-network. As the paper shows, it is possible to think of this as a new place in applications infrastructure which didn't exist before. It's an open, in-traffic service processor which allows application vendors to place code in-stream of the network and do real-time monitoring, application QOS, event correlation, business intelligence, transparent application integration, and many use cases which have not been discussed here.

One particularly interesting use case is in Utility Computing. For example, "intermediary code" can be placed in aggregation locations in a datacenter network, which can transparently intercept and proxy network traffic at message flow layers and redirect them to the utility. In this way intra-cluster server to server traffic, message queue traffic, remote file system traffic, and even client/server database traffic can be transparently integrated to the cloud. On top of this capability, because we are in-network, full control of bandwidth management, per-message flow SLAs, usage based billing models, monitoring and management, are all available as possibilities. For utility computing vendors, this is potentially a way to seamlessly offer cloud computing services into the enterprise.

To recap the details of how this works, this place in application infrastructure presents reconstructed message streams (not raw packet streams) which the application code has requested, for example, specific to a type of application or protocol. This is done transparently in the network, like a firewall, at wire speed, using deep packet and deep message inspection technology. Message traffic is extracted according to a semantic definition of the actual application protocol, a behavioral definition of the protocol, or even content matching using Regex or XPath.

This message stream is served up to the in-network code, which, using a new streaming programming model, does whatever new in-network application logic the application designer wishes, which may involve the

applications servers, and then delivers the potentially modified stream back into the network – invisibly to the endpoints. Along the way the application code could have copied, blocked, modified, time stamped, changed network parameters such as QOS, etc, of this message stream. In this way generalized applications visibility and control are achieved paving the way for applications virtualization, transformation, SLAs, in a place in the topology where servers don't have access. It is a natural extension of the distributed programming model into the network, not replacing databases or servers, but adding new capabilities.

We believe there are many, many interesting uses for this new location in applications infrastructure that we can't envision, but utility computing and applications developers will. From a business perspective, this enables new footprint for applications code and new value propositions.

As to the programming model and API, we are working towards an open, published streaming Java and then C/C++ programming model development. Ultimately we do not see this as a proprietary Cisco technology. We see this as a published standard.

9. References

- [1] Clark, D. D., Jacobsen, V., Romkey, J., and Salwen, H. "An Analysis of TCP Processing Overhead" *IEEE Communications Magazine* 6, (June 1989), 23--29
- [2] George Varghese. "Trading packet headers for packet processing" *Technical Report WU9416*, Dept. of Computer Science, Washington University, 1994
- [3] Youg H. Cho, Shiva Navab, and William Mangione-Smith, "Specialized Hardware for Deep Network Packet Filtering", *International Conference on Field Programmable Logic and Applications (FPL)*, Montpellier France, Sep, 2002
- [4] Ni, Jia; Lin, Chuang; Chen, Zhen; Ungsunan, Peter; „A Fast Multi-pattern Matching Algorithm for Deep Packet Inspection on a Network Processor" *International Conference on Parallel Processing, 2007, ICPP 2007*. 10-14 Sept. 2007 Page(s):16 – 16
- [5] LakshmiPriya, T.K.S. Hari Prasad, V. Kannan, D. Singaram, L.K. Madhan, G. Sundaram, R.M. Prasad, R.M. Parthasarathi, R. Anna Univ., Chennai; "Evaluating the Network Processor Architecture for Application-Awareness" *2nd International Conference on Communication Systems Software and Middleware, 2007. COMSWARE 2007*, Publication Date: 7-12 Jan. 2007
- [6] Suntae Hwang; Naksoo Jung; "Dynamic scheduling of Web server cluster", *Ninth International Conference on Parallel and Distributed Systems*, 2002. Proceedings, 17-20 Dec. 2002 Page(s):563 – 568
- [7] Madhukar, A. Williamson, C. "A Longitudinal Study of P2P Traffic Classification", *14th IEEE International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems, MASCOTS 2006*. Publication Date: 11-14 Sept. 2006
- [8] <http://www.opsec.com/>

- [9] <http://www.3com.com/osn/>
- [10] <http://devcentral.f5.com/>
- [11] <http://www.juniper.net/partners/osdp.html>
- [12] Abstract Syntax Notation One (ASN.1) Specification of Basic Notation ITU-T Rec. X.680 (2002) | ISO/IEC 8824-1:2002, <http://www.itu.int/ITU-T/asn1/>
- [13] Tim Bray, Jean Paoli, C. M. Sperberg-McQueen, Eve Maler, François Yergeau, John Cowan, ed. "XML 1.1 (Second Edition)", *W3C Recommendation*, 16 August 2006, <http://www.w3.org/XML/>
- [14] See both <http://www.ws-i.org/deliverables/index.aspx> and <http://www.oasis-open.org/specs/index.php> for current reference specifications of Web Services (too many to list).
- [15] James Clark, Steve DeRose, ed., "XML Path Language (XPath) Version 1.0" *W3C Recommendation* 16 November 1999, <http://www.w3.org/TR/xpath/>
- [16] "Document Object Model (DOM) Technical Reports", W3C, <http://www.w3.org/DOM/DOMTR>
- [17] Yanlei Diao, Mehmet Altinel, Michael J. Franklin, Hao Zhang, and Peter Fischer. „Path Sharing and Predicate Evaluation for High-Performance XML Filtering”. *TODS*, December 2003.
- [18] Yanlei Diao, and Michael J. Franklin. "High-Performance XML Filtering: An Overview of YFilter". *IEEE Data Engineering Bulletin*, March, 2003
- [19] Yanlei Diao, Peter Fischer, Michael Franklin, and Raymond To. "YFilter: Efficient and Scalable Filtering of XML Documents" *Demo paper, in Proceedings of ICDE 2002*, February 2002
- [20] D. Mills, "RFC 1305 Network Time Protocol (Version 3) Specification, Implementation and Analysis." *IETF* March 1992
- [21] IEEE Std. 1588-2004 "Precision clock synchronization protocol for networked measurement and control systems", 2004, *TC9-Technical Committee on Sensor Technology of the IEEE I&M Society*
- [22] Eidson, John; Garner, Geoffrey M.; Mackay, John; Skendzic, Veselin; "Provision of Precise Timing via IEEE 1588 Application Interfaces", *IEEE International Symposium on Precision Clock Synchronization for Measurement, Control and Communication, 2007*. ISPCS 2007. 1-3 Oct. 2007 Page(s):1 – 6
- [23] Lee, Kang; Song, Eugene; "Object-oriented Model for IEEE 1588 Standard", *IEEE International Symposium on Precision Clock Synchronization for Measurement, Control and Communication, 2007*. ISPCS 2007., 1-3 Oct. 2007 Page(s): 7 – 12
- [24] 802.1Q-2003 "IEEE standards for local and metropolitan area networks. Virtual bridged local area networks", *LAN/MAN Standards Committee IEEE Computer Society, USA*, Publication Date: 2003
- [25] 802.11-2007, "IEEE Standard for Information technology-Telecommunications and information exchange between systems-Local and metropolitan area networks-Specific requirements - Part 11: Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) Specifications", *LAN/MAN Standards Committee IEEE Computer Society, USA*, Publication Date: June 12 2007
- [26] E. Rosen, Y. Rekhter. "RFC 2547 BGP/MPLS VPNs." *IETF*, March 1999
- [27] D. Awduche, J. Malcolm, J. Agogbua, M. O'Dell, J. McManus "RFC 2702 Requirements for Traffic Engineering Over MPLS", *IETF*, September 1999
- [28] Rosen, E.. A. Viswanathan, R. Callon "RFC 3031 Multiprotocol Label Switching Architecture", *IETF*, January 2001
- [29] J. Quittek, T. Zseby, B. Claise, S. Zander. "RFC 3917 Requirements for IP Flow Information Export (IPFIX).", *IETF*, October 2004
- [30] S. Leinen "RFC 3955 Evaluation of Candidate Protocols for IP Flow Information Export (IPFIX)", *IETF*, October 2004
- [31] B. Claise, Ed.. "RFC 5101 Specification of the IP Flow Information Export (IPFIX) Protocol for the Exchange of IP Traffic Flow Information", *IETF*, January 2008
- [32] J. Quittek, S. Bryant, B. Claise, P. Aitken, J. Meyer, "RFC 5102 Information Model for IP Flow Information Export.", *IETF*, January 2008
- [33] "JSR-000003 Java™ Management Extensions (JMX) Specification", <http://jcp.org/aboutJava/communityprocess/final/jsr003/>