Influence of friction between layers of mechanically lined pipes on collapse resistance

Moving Applications from the Host to the Network: Experiences, Challenges and Findings

Ivano Cerrato, Marco Pramotton and Fulvio Risso
Department of Control and Computer Engineering, Politecnico di Torino
Corso Duca degli Abruzzi 24, 10129 Torino, Italy
E-mail: {name.surname}@polito.it

***Abstract*—Some recent works propose to extend network devices (e.g., routers) with the possibility to execute additional user-provisioned software operating on the data-plane. This enables network devices to be enriched with new functionalities, potentially decided at run-time directly by the end users. This paper focuses on one of such programmable routing platform and presents our experience in developing new software (namely, a parental control service) in that environment. In addition, we describe also two extensions to our platform that were needed to accommodate the necessity of our applications.**

I. Introduction

The *end-to-end principle*, one of the pillars of the original Internet, is increasingly challenged by the presence of many middleboxes that manipulate network traffic, such as for security or traffic optimization purposes. In fact, a recent study [1] demonstrates that the applications that can modify the traffic in transit (such as application-layer firewalls, intrusion detection systems, transparent caching, WAN accelerators, and more) are quite common in many network paths.

In a recent work [2] we proposed a prototype of edge router that allows to customize the *data-plane* of the network, by allowing third parties (e.g., end users, application service providers, network service providers) to install their own applications operating on the packets traversing the router itself. This way the router is no longer limited to *forward* the traffic, but it can inspect and potentially *modify* all the packets in transit and it can export a sort of dedicated execution environment to the applications that require those features. This model could simplify the network operations, as the dedicated middleboxes mentioned above could become a set of software images installed on the network edge router.

This paper aims at validating the potentialities of our routing platform from the point of view of the final developer and it presents our experience in developing a complex application-layer service. First, it describes the programming architecture of our prototype, showing the main functions offered to the programmer and a brief overview of the API. Second, it presents our experience in implementing a complex service on our architecture, namely a parental control, and shows a preliminary characterization of its performance. The parental control was chosen because it represents a service that can take many advantages from the possibility to be executed in the network instead of in user devices. In fact, by being executed on an edge router, the parental control is able to inspect all the traffic to/from the users that need to be protected (e.g., kids), regardless of the device their are using to connect to the Internet, as well as their access network (e.g., domestic WLAN, 4G, etc.). Furthermore, our prototype will allow to safely share the same physical device (e.g., a tablet) among many users (e.g., kids, parents), as the network is able to recognize the users and install the proper applications operating on their traffic.

This paper is structured as follows. Section II presents the related work, focusing on both programmable routers and parental control services. Section III summarizes the main concepts of our programmable router and it presents the new extension that were needed to implement the service we had in mind. Section IV presents the programming architecture and an overview of the exported API, while Section V describes our parental control service. Finally, Section VI shows some numbers that come from the deployment of the parental control service and Section VII draws some conclusive remarks.

II. Related Work

This work is based on the programmable router presented in [2] and briefly summarized in Section III, which enables third parties to install their own applications operating on the *data-plane* of the node itself. The idea of a programmable network router has been explored in many papers; perhaps [3] represents the closest idea to our proposal. In fact, [3] proposes a minimal network operating system based on the Click modular router software [4], but is more oriented to network providers and it does not allow applications coming from end users to be deployed on the device. In our case, we emphasize the possibility for end-users to relocate their data-plane applications from user terminals to the network, so that applications themselves can operate independently of the physical device (and its operating system) in use; furthermore, those applications do not consume memory and CPU cycles on the user terminals, which favors particularly mobile devices with strict constraints in terms of power consumption.

To validate the programmable router, we chose to develop a parental control service; even if many implementations of this application already exist (installed on user terminals, but also deployed within the network, i.e., on routers and proxy servers), they all suffer of several limitations. Applications such as *Net Nanny* [5], *safeeyes* [6] and *Davide.it* [7] are installed on host devices, and they may not be able to properly

protect kids because of the many different devices owned by each person. In fact, parental controls could operate in different ways and/or offer different degrees of protection on different devices and, in some cases, they may not even exist on some platforms. Among the router-based solutions, we can cite manufacturers like Cisco [8] and Netgear [9], which offer parental control services running on their devices. However, these applications are poorly customizable and implement only the features decided by the manufacturers themselves. Users cannot upgrade the service with new features and have to wait for the manufacturer to implement them. Other solutions are based on a proxy server, such as *DansGuardian* [10]. In this case the web browser on the user terminal must be configured to send all requests to the proxy where the parental control application is running, rather than directly to the destination web server. However, this configuration on the client terminal can be easily bypassed by reconfiguring the application in order to access the Internet directly. Furthermore, this solution is limited to the protection from threats coming from web traffic. Finally, we can cite services that filter DNS requests (e.g., *OpenDNS* [11]), which cannot block applications that do not use the DNS such as instant messaging and others.

We believe that our parental control service installed on the edge router has many advantages compared to previous solutions. First, it is able to protect kids regardless of the physical device they use to connect to the Internet, as well as to recognize the user connected to the network and act differently according to his profile. Second, it enables a fine tuning of the service by allowing end users to install the applications they want, e.g., with additional or more advanced capabilities. Third, it is able to protect children independently from the applications they use, as it operates on all the network traffic. Finally, it has the potential to protect minors independently from the location they connect to the Internet. In fact, our system requires the user to go through an authentication phase before being able to connect to the network, which allows the system to detect the user identity and to install exactly the applications associated to that user.

III. The Programmable Edge Router

This section summarizes the main concepts of the programmable edge router [2] and presents the new extensions related to the remote execution environment and the storage service. The main modules of the router are shown in Figure 1, which also provides an overall view of the entire system.

In our router the network is split in multiple slices, each one associated with a different user. This way, a user can install and manage data-plane applications operating only on his slice, i.e., on packets coming from *or* directed to its MAC address, without impacting on the services requested by other entities. Each slice is mapped to a different Private Execution Environment (PEX), a sort of virtual machine dedicated to the specific user. This execution environment exports a set of functions (detailed in Section IV) that can be used to build applications (written in Java), hence enabling them to operate on the traffic of that user. In fact, those applications will be called sequentially on user's packets; if an application modifies the content of a packet, all the following ones will receive the modified payload. Each user is enabled to install his preferred applications on his slice and to decide the calling order. Furthermore, some users (e.g., parents) have the additional privilege to install hidden applications on other slices (e.g., the ones of their kids), in order to guarantee some specific services on the traffic of those users. Finally, a new PEX is created when the router detects a new user connected to it, upon redirecting his traffic to a captive portal and asking the user to provide his credentials.

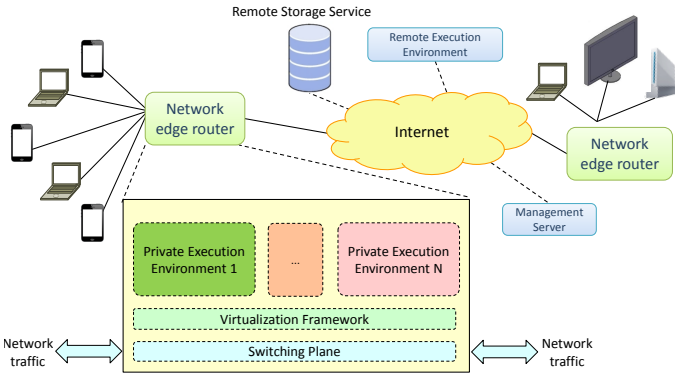

Fig. 1. Overview of the entire system.

The logical architecture of the router, depicted in Figure 1, includes three main components. The **Private Execution Environment** executes the user applications and receives/sends the traffic from/to the **Virtualization Framework**. This component is in charge of virtualizing the router, which enables the PEX to operate such as it is the only container running on the node itself. For instance, the Virtualization Framework enables the slicing and redirects the incoming traffic to the PEX of all the slices the packet belongs to. Finally, the **Switching Plane** is in charge of sending/receiving packets to/from the Virtualization Framework, and of handling the transmission of the traffic to/from the physical ports of the router. This component may be replaced by a real router connected to the other components through an OpenFlow [12] connection.

Moreover, a **Management Server** exists that is in charge of keeping the user database and handling a part of the authentication process. In general, this entity coordinates the entire set of edge routers; in fact, it contains also the list of applications to be installed, together with other information needed to manage the system.

When we started writing some more complete applications for our router, we recognized that the architecture presented so far was missing some important features. For instance, we felt we needed at least a basic version of a *remote storage service* and a *remote execution environment*, whose current implementations are detailed in the following.

A. Storage Service

One of the applications we developed for our parental control service is a *DNSFilter* module (detailed in Section V),

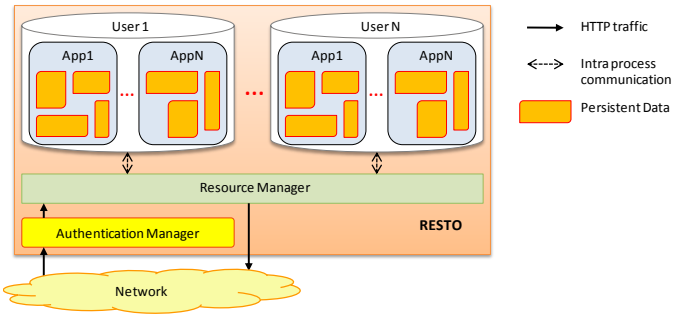

Fig. 2. Internal view of the Storage Service.

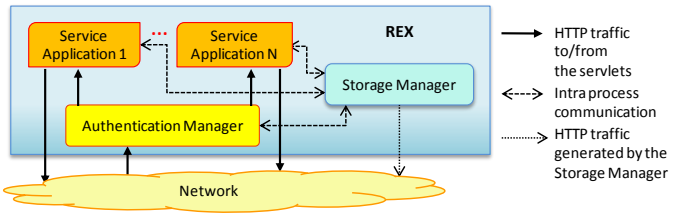

Fig. 3. Exploded view of a REX.

which basically filters DNS requests and rejects those that refer to names included in a forbidden list. While at the beginning we included the blacklist in the application itself, we recognized that it was much better to store that information on a remote storage. In general, applications may need to store persistent data, such as state information or configuration parameters that must be preserved across multiple execution of the same service. In the current system, the applications running on the router had only the "volatile" storage provided by the variables that are defined in the application itself.

As a consequence, we added a **Remote Storage Service** (RESTO), whose internal architecture is depicted in Figure 2. The Storage Service includes the **Authentication Manager**, i.e., the component that authenticates the couple user/application (more details in Section III-C) and the **Resource Manager**, which is responsible for reading/writing data upon requests from applications, as well as for deleting information when its owner is uninstalled. The RESTO service organizes the data in a tree based on the *application* that generated it and on the *user* who is executing that application in his own PEX. Vice versa, multiple instances of the same application running on different PEXs associated with the same user (e.g., a user connected to the network through a smartphone and a laptop) share the same data; the implementation of different storage areas for those instances are under the responsibility of the programmer. The "remote" characteristic of the service enables applications to access their persistent data independently from the physical router they are running on, and it enables that data to be accessed also from other parties (e.g., other services residing in the cloud).

The RESTO module is deeply integrated with the rest of the platform and it allows programmers to read/write data with simple primitives, while other issues (such as the authentication process, which guarantees that a user can write/read only his data) are under the responsibility of the system and are transparent to the programmer. The system takes also care of cleaning up the data associated to a given application/user when that application is removed from the user and hence does not longer belong to the PEX associated to his slice. It is worth noting that we do not force programmers to exploit the RESTO; in fact, they are still enabled to save their data wherever they want. This way, however, they must address by themselves all the issues already solved by our platform.

B. Remote Execution Environment

The *DNSFilter* module mentioned before showed another problem. As the list of forbidden sites was rather large and was consuming a huge amount of memory on the router, it would be better to split the application into a router part, with a short list, and a server part, which keeps the full list and that is invoked upon demand. This suggested us that there may be a class of applications that can be split in multiple portions hosted on different locations, such as a part running on the router and another running on a remote server.

While, in line of principle, the programmer can implement this splitting by setting up a remote application service and modify the application running on the router in order to access to that service, we decided to offer him another possibility that looks more integrated with our solution. Then, we defined a **Remote Execution Environment** (REX), a module that can host the "server" part of applications and that can offer some standard functions to the programmer. This module, whose architecture is shown in Figure 3, provides a Java-based execution environment very similar to the PEX, and also the exported API has similarities with the one present in that module. For instance, obviously no primitives are available for reading/modifying network packets, but others (e.g., the API for accessing the RESTO service) are the same in both the REX and PEX environments.

Finally, the REX programming environment hides both the communication between the code executed on the router and the one running in the remote environment, and all the authentication/authorization issues.

C. Communication and authentication

Programmers access the REX and the RESTO through an API that hides both the details of the communication (which occurs through HTTP) and the authentication process. With respect to the authentication, the Management Server randomly generates a secret key when the user logs in, which is shared between the user's PEX, RESTO and REX services. This secret allows the remote components to identify the user and the application when they receive a request from the PEX.

In fact, we insert in the HTTP requests information such as the *username* of the user who is running the application that requested the service, the *application name* identifying the application that makes the request, a PEX identifier that uniquely identifies the PEX in which the application

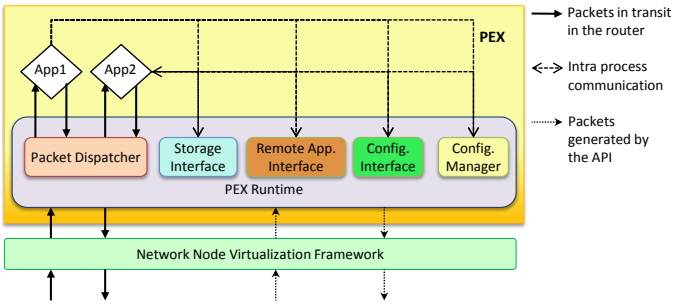

Fig. 4. Exploded view of a PEX hosting two applications.

is running, the *router identifier* that uniquely identifies the application as being executed on a given router. Finally, part of those information and the secret key mentioned before are given as input to the `sha-256` algorithm in order to generate an unique signature that will be used by the remote party as authentication key.

It is worth noting that all these parameters are under the control of the PEX and are not visible by the application. As we suppose that the PEX is trusted (while the user-provider code running in it may not), we can safely assume that those parameters are enough to guarantee the proper interaction with the remote services, at least in our prototype. In fact, when an HTTP message reaches the remote service, the authentication module is able to recognize the user / application / instance of application that asked for the service. Then, it forwards the request to the proper service handler, being it a storage module or a service in the REX.

IV. Programming the PEX

This section describes the programming architecture of the PEX, by detailing the various components depicted in Figure 4 and by providing an overview of the exported API.

A. Callbacks

As the PEX exports an event-driven programming model, an application is requested to implement a set of callbacks that are called when specific events occur. In particular, `OnStartUp` and `OnShutDown` must contain the code that has to be executed when the application is started/stopped as a consequence of user's commands. `OnReceivedPacket` is instead called when a new packet is available in the system and needs to be processed. In this case, the program receives the packet and a set of metadata such as the physical port of the router on which that packet was received. This method must return `DROP` or `CONTINUE`, depending on whether the packet must be discarded or it can be forwarded to the next recipient, which can be either the application that follows in the same PEX or the Virtualization Framework, in case that the application was the last one in that PEX.

B. PEX Runtime

The PEX runtime creates the environment on which applications are executed, and it is derived from the Beacon OpenFlow controller [13]. It includes several modules that can be exploited by applications through the proper API (e.g., the interface toward the REX/RESTO services), the packet dispatcher and the interface toward the Virtualization Framework. Particularly, the latter receives the packets encapsulated in the OpenFlow protocol from the network and it sends the raw data (without OpenFlow headers) to the user applications. The opposite process is implemented when packets have to be returned back to the network. Finally, the **Configuration Manager** implements a set of REST services that enable users to *(i)* install/uninstall applications, *(ii)* change their calling order and *(iii)* start/stop applications already installed.

C. Packet Dispatcher

The **Packet Dispatcher** is the component in charge of delivering packets to the applications by calling their `OnReceivePacket` handler each time that a new packet reaches the PEX. This module can dynamically add/remove new applications in the calling stack of the PEX. As applications can be associated to different privileges (e.g., packets can be received in read-only mode), the `GetPrivileges` method allows them to know their privileges and act accordingly. Since an application may ignore this information and perform illegal actions, this module also implements techniques to ensure that privileges cannot be violated. Finally, the Packet Dispatcher exports the `RegisterFilter` and `UnregisterFilter` methods, which enable applications to receive only the packets matching a given filter (e.g., HTTP traffic); however, an efficient implementation of this function is left to future work.

D. Storage Interface

The **Storage Interface** implements the communication to the RESTO mainly through the intuitive `SaveData` and `ReadData` methods. Data to be stored must be Java objects implementing the interface `Serializable`, or also Java primitive data types such as `int` and `float`; this way, the system is able to manage any kind of data, even objects defined by programmers. Optionally, data can be stored with some additional metadata such as the timestamp in which the data was modified, and the responsible of that change (in terms of PEX identifier and Router identifier). In order to manage the concurrent access to data, the Storage Manager also exports the `LockResource` and `UnlockResource` methods, which can be used to implement atomic modification on that data even in presence of multiple running applications associated to that user, e.g., in case multiple terminals associated to the same user are connected to the network. Finally, this interface implements all the mechanisms required for remotely authenticate the user on the RESTO, as explained in Section III-C.

E. Remote Application Interface

Similarly to the previous component, the **Remote Application Interface** handles the interaction with services hosted in the REX. This module exports the simple `Get`, `Post`, `Put` and `Delete` methods, which derive from the HTTP

methods defined in a REST interface. Those methods create the appropriate HTTP request for the resource specified as a parameter and return to the application the HTTP response coming from the remote service. The remote URL is partially created automatically by the PEX (e.g., the application name), while other information are set by the application (e.g., the part identifying the resource and the additional parameters that may be needed). Also in this case this component hides the entire authentication process to the applications.

F. Application Management Interface

The **Application Management Interface** enables each application to implement the primitives that can be used to configure or monitor the service from the external world. In fact, each application can be reached by the slice owner by typing the standard URL `http://config.ctrl/application_name/`. The system will check the request for permissions, then redirects it to the REST services exported by applications. It is worth noting that both the semantic and the syntax of the data exchange is completely application-dependent.

V. Parental control

The parental control service developed on top of our programmable platform consists of the following applications.

GSafe exploits the *Google safe search* [14] feature of the Google search engine to filter harmful contents. When active, GSafe enables the safe search by changing the URL in all the HTTP `GET` messages towards Google and related to a search. In particular, the URL is extended with `safe=active` or `safe=strict`, depending on the selected level of protection specified in the configuration parameters. Currently, the implementation can operate only on packets that (after the modification) do not exceed the MTU. We plan to introduce a stream reassembly function in the API in the future, in order to enable programmers to avoid this issue and then to allow them to operate also on *messages* in addition to *packets*.

DNSFilter is an application that prevents children from reaching disturbing websites whose URLs are included into a blacklist. As URLs are organized by topic (e.g., porn, drug, etc.), a configuration parameter (visible to parents) can be used to enable/disable one or more sections. We implement this application both in a *monolithic* version, entirely running on the PEX, and in a *split* version in which DNS packets are received by the application on the router, checked against a small cache, and in case of miss the request is redirected to a remote service on the REX. Based on the result of the check, the application can drop the DNS message, or let it go on its way. In addition, DNSFilter gathers all the DNS names translated, in order to allow parents, through the web interface, to inspect which sites were accessed by their children.

TimePeriod can block the access to the Internet during a given time slot, as well as it can limit the amount of time for which kids can be logged in into the system. For instance, a child could be enabled to surf the network only from 2 pm to 9 pm; in addition, he could be allowed to spend no more than two hours per day on the Internet. Similarly to the previous applications, TimePeriod exports a web interface that enables parents to configure the application itself.

Finally, **SkypeBlocker** is able to identify and discard the Skype traffic according to the signature defined in `nDPI` [15].

All these applications exploit the RESTO in order to store the configuration parameters selected by the parents.

VI. Validation

This section validates our platform and parental control service through different categories of test. The test set up consists in a Fast Ethernet network that includes a user laptop directly connected the programmable router; a set of servers implementing the DNS server, RESTO, REX and the management server are directly connected to the router. All the servers are workstations with an Intel Core2 processor (Q8400 at 2.66MHz), the router runs an Intel i5 3450S at 2.8GHz, while the laptop is a Intel Core2 P8700 at 2.53 GHz. All the machines have 4GB RAM and a 7200rpm hard disk in the range 250-320 GBytes; moreover, they were preloaded with the Linux Debian 7 operating system running at 64 bits.

A. Starting the PEX

This test measures the time required to activate the PEX environment upon the receipt of a successful login from a new user. When the system authenticates the user through the captive portal, it starts a new PEX on that edge router and it activates all the applications associated to that user. This process includes several steps summarized in the top part of Figure 5, such as the time required to check the user credential in the management server, the time needed to configure the environment to host a new PEX, the time needed to start a new PEX with all the requested applications, and finally the time needed to create the slice for the user and map this to the newly created PEX. Although we agree that our implementation can be improved, everything completes in less than 5 seconds in our operating conditions. Particularly, the time required to start each application (which requires downloading it from the management server, injecting it in the existing PEX, and starting it) is negligible compared to that needed to completely activate the execution environment. In fact, the worst time we get refers to the DNSFilter application, which completes this process in 195 ms. Furthermore, many tasks such as installing applications are launched in parallel, contributing to keep the overall duration low.

B. Accessing the RESTO

This test shows the latency introduced when an application uses the RESTO. In order to obtain this time, we wrote a simple program that saves and reads resources of different sizes to/from the storage service each time a packet is received. Our numbers take into account the worst condition as we read resources that were not in the cache of the RESTO.

The application was repeated thousand times and the numbers were averaged in order to obtain the results shown in Table I. As expected, readings are always slower than

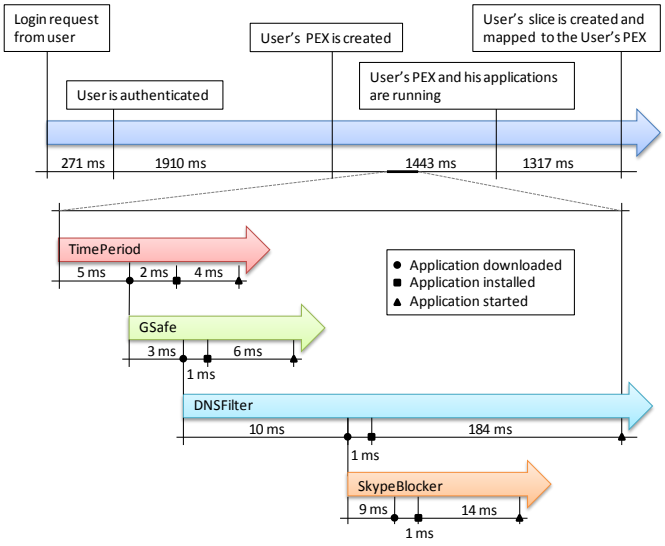

Fig. 5. Starting a PEX with four applications.

TABLE I
LATENCY IN ACCESSING THE RESTO.

	10B	**100B**	**1KB**	**10KB**	**100KB**
write [ms]	3.49	4.27	6.15	17.25	129.61
read [ms]	2,29	3,24	4,29	14,24	115,59

writings, and the latency grows with the size of the managed resource. Numbers confirm also that the RESTO service is most appropriate for applications that need occasional access (e.g., to store/load configuration parameters), while it may not be appropriate for an application that requires an access to this service each time a new packet is received.

C. Exploiting the REX

This test evaluates the impact of the REX in terms of memory saving on the edge router and of latency. For this purpose, we run both the monolithic and the split flavors of the DNSFilter.

Since the split application does no longer use the black-list within the PEX, the memory consumption at the router decreases considerably: 141MB with the monolithic version, against 24MB with the split application. On the other hand, time needed to serve the DNS request increased from 1.4ms to 7.8ms, due to the additional steps (e.g., creating and sending the HTTP request, etc.) required to obtain the answer. These results were obtained by averaging numbers coming from thousand queries toward the local DNS server set up in our network, which was configured to answer to all the queries without forwarding them to the Internet, hence avoiding any issue not under our control.

VII. CONCLUSIONS

This paper presents our experience in developing a parental control service for the programmable router we presented in [2]. This work allowed to test, with the eyes of the final developer, the validity of our platform. In fact, we felt the necessity to extend the programmable environment in order to accommodate some additional requirements of our application, which have not been foreseen in our original prototype. In particular, we added the *Remote Storage Service* (RESTO), which enables applications to save their persistent data, i.e., information that must be maintained among different executions of the applications themselves. Experimental results shows that an occasionally access to the RESTO (e.g., reading/writing configuration parameters) does not significantly reduce the performance of applications. We also defined the *Remote Execution Environment (REX)*, which enables applications to exploit a remote service on the web for their purposes. Thanks to the REX, we were able to partition applications into an "edge" and a "cloud" portion, hence reducing the hardware requirements on the edge node (e.g., in terms of memory consumption), although at the cost of introducing some additional latency in the applications.

In future, we plan to extend the services offered by our platform, and to improve the API it exports to application developers. New services may include, among the others, the support for encrypted traffic and the transparent handling of application-level messages. This way, programmers can focus on the logic of their applications, instead of on these low level (and tedious), but often necessary, tasks.

REFERENCES

[1] M. Honda, Y. Nishida, C. Raiciu, A. Greenhalgh, M. Handley, and H. Tokuda, "Is it still possible to extend tcp?" in *Proceedings of the 2011 ACM SIGCOMM conference on Internet measurement conference*, ser. IMC '11. New York, NY, USA: ACM, 2011, pp. 181–194.

[2] F. Risso and I. Cerrato, "Customizing data-plane processing in edge routers," in *Proceedings of the European Workshop on Software Defined Networking (EWSDN)*, 2012, pp. 114–120.

[3] M. Ahmed, F. Huici, and A. Jahanpanah, "Enabling dynamic network processing with clickos," in *SIGCOMM*, 2012, pp. 293–294.

[4] R. Morris, E. Kohler, J. Jannotti, and M. F. Kaashoek, "The click modular router," in *Proceedings of the seventeenth ACM symposium on Operating systems principles*, ser. SOSP '99, 1999, pp. 217–231.

[5] ContentWatch. Net nanny. Stable release: 6.5 (Windows) 2.0 (Mac). [Online]. Available: http://www.netnanny.com

[6] McAfee. safeeyes. [Online]. Available: http://www.internetsafety.com/safe-eyes-parental-control-software-affiliate.php

[7] Davide.it. [Online]. Available: http://www.davide.it/

[8] Cisco systems. [Online]. Available: http://homesupport.cisco.com/en-us/support/ccc/PARENTALCONTROLS

[9] Netgear. Live parental controls. [Online]. Available: http://www.netgear.com/lpc

[10] D. Barron. (2011, Aug) Dansguardian. Stable release 2.12.0.0. [Online]. Available: http://dansguardian.org

[11] Opendns. [Online]. Available: http://www.opendns.com/parental-controls

[12] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner, "Openflow: enabling innovation in campus networks," *SIGCOMM Comput. Commun. Rev.*, vol. 38, no. 2, pp. 69–74, Mar. 2008.

[13] D. Erickson. (2012, Apr) The beacon openflow controller. [Online]. Available: http://www.beaconcontroller.net

[14] Google. Google safe search. [Online]. Available: http://support.google.com/websearch/bin/answer.py?hl=en&answer=510

[15] (2012, Apr) ndpi. [Online]. Available: http://www.ntop.org/products/ndpi/